

**Master thesis**  
**Computing Science**  
**Cyber Security Specialization**



Radboud University

---

**Receiver anonymity within a  
distributed file sharing protocol**

---

*Author:*  
*Solo Schekermans*  
*S4811070*

*Main supervisor/assessor:*  
*Dr. Jaap-Henk Hoepman*

*Second assessor:*  
*Dr. ir. Erik Poll*

August 2019

**Abstract.** In this thesis, we explore the field of receiver anonymity within a file sharing protocol. Exploring existing solutions, such as Resilio Sync, reveals that at this moment, a “ready to use” solution does not yet exist. Looking at the most promising solution that offers a private manner of sharing files over a peer-to-peer network, we find that the most suitable protocol to implement receiver anonymity is actually BitTorrent.

An existing solution, called BitBlender, does a good job of adding receiver anonymity to BitTorrent by using crowds, but does have several flaws that need to be resolved. Most flaws are easy to mitigate but one, the absence of incentives for crowd members, requires an overhaul of the protocol.

To incentivize the crowd members, a token based protocol is proposed that utilizes blind signatures and attribute based credentials.

**Keywords:** Anonymity, peer-to-peer, file sharing, BitTorrent, Resilio Sync, BitBlender, incentivizing relay peers, blind signatures, attribute based credentials.

## Contents

1	Introduction	4
2	Contextual environment	5
2.1	Definitions	5
2.2	Architecture models	6
2.3	Existing solutions/products	7
2.4	Privacy	8
3	Goal of the system	10
4	BitTorrent & Resilio Sync	12
4.1	BitTorrent Protocol	13
4.2	Resilio Sync protocol	21
5	BitBlender by Bauer et al.	26
5.1	Flaws with BitBlender.	29
6	Solutions to the BitBlender flaws.	31
6.1	Providing an incentive for relay peers.	32
6.2	Technical implementation for incentivizing relay nodes.	35
7	Conclusion	44
8	References	45
9	Appendix I: Existing solutions	47
9.1	Testing framework	47
9.2	Existing solutions	48
9.3	Peer-to-peer model	57
9.4	Self-hosted model	59
9.5	Conclusion	64
10	Appendix II: Token typing with encrypted strings in partially blinded signatures	65

## 1 Introduction

Within file sharing we can distinguish three entities. First, the “sender” is an actor who has a file and wants to share it with others. Second, the “file” is the abstract name we will use in this thesis to represent what a sender is able to send, so files, folders, archives and such are all referred to as a “file”. And third, the “receiver” is an actor who wants to obtain the file anonymously.

Imagine a scenario in which three actors exist: Alice, Bob and Charlie. Alice has a file that Bob and Charlie wish to have. Alice and Bob know and trust each other, but Charlie represents an unknown entity. Charlie wants to obtain the file without a third party (passive observer) knowing he obtained it. Alice, Bob and Charlie are geographically separated by long distances and therefore want to utilize the internet.

The above scenario is an example where receiver anonymity within an anonymous file sharing protocol comes into play. The goal of this thesis is to research whether it is possible to design a protocol in which receiver anonymity is provided. To achieve this, research has been done into what existing solutions offer and if there are any protocols that can be expanded with receiver anonymity.

An overview of existing solutions and their properties appears in appendix I. We assessed each one for their potential to offer receiver anonymity. Based on this assessment we conclude that peer-to-peer systems offer the most potential, especially BitTorrent extended with BitBlender.

Section 2 will start with defining the contextual environment, starting off with clearing up some definitions followed by giving the possible architectural models in which the file sharing system can be made and finishing with what privacy we are actually trying to achieve. Section 3 contains the design goals for this system. Section 4 researches BitTorrent and Resilio sync. BitTorrent is a popular peer-to-peer file sharing protocol. Resilio Sync is an existing product that leverages the peer-to-peer model to offer private file synchronization and sharing. Section 5 researches what BitBlender is and what flaws it contains. BitBlender is an extension to the BitTorrent protocol that adds anonymity using crowds. Section 6 proposes several solutions for the flaws found in BitBlender with the biggest being adding an incentive for relay peers. Finally, section 7 contains the conclusion.

## 2 Contextual environment

To define what our protocol should be capable of, we will compose a set of design goals. To get a proper set of design goals we first define the contextual environment that our protocol may operate within. To prevent any misunderstandings about what the file sharing means exactly, a definition will be given for file backup, file synchronization and file sharing.

Creating an anonymous file sharing system can be achieved by using one of the following two architectures: the client-server model and the peer-to-peer model. We take a look into how those architectures work, what their drawbacks are and what their benefits are. To sketch a complete picture, we also look into a third architecture which is the self-hosted model. The self-hosted model is essentially one of the other two architectures in a self-hosted/controlled/owned environment.

Thereafter, we will investigate existing solutions to learn how they work, which architecture they use and possible if a solution that offers receiver anonymity.

Finally we will look into what privacy we expect from the system, and define a clear privacy goal to achieve.

### 2.1 Definitions

To prevent conflicting understandings of the actions under consideration we give a clear definition of what the actions entail exactly.

#### **Backup.**

A backup of data is defined as a second copy of a file on a different location. If the original is deleted from the source, it will not be deleted on the backup location. Backups are useful to combat the effects of data corruption, viruses, hardware issues and more. Versioning<sup>1</sup> is not in the core functionality of backup, but a useful addition. This means keeping older versions of a file when a newer version is written to the backup. Versioning comes in handy when a user accidentally deletes all contents of a file, but not the file itself. An older version contains the deleted content and can be retrieved through versioning, whilst without versioning only the new version could be downloaded which would be empty. There are a couple of backup types that can be used, for example: full (a complete copy each time), incremental (only back up changed data since previous backup) and differential (only backup changed data since first backup)<sup>2</sup>.

#### **Synchronization.**

Synchronization is not the same as a backup. It keeps the content of two or more locations identical. So, a change in one location means a change in all other locations.

---

<sup>1</sup> Not to be confused with “Software Versioning” which means giving version numbers to releases.

<sup>2</sup> For information on backup types see: <https://searchdatabackup.techtarget.com/tip/Data-backup-types-explained-Full-incremental-differential-and-incremental-forever-backup>

Unlike a backup, if a file is deleted anywhere, it will be deleted everywhere. Synchronization enables users to have all their files over multiple devices, in a way it can be used as a backup (if one location is corrupted other locations may still have the files) but this is not advisable or the goal of synchronization.

### **Sharing.**

Sharing is substantially different from backups and syncing. It allows one or more clients to access data. This can be done in many ways. For example: sending a file through an email attachment, placing a file on an accessible web server or handing someone a USB drive with a file on it. These are all forms of file sharing. For this thesis we will define file sharing as making some data available for download. This means we do not need to know a client's address to send a file to. We only need to relay the file location to the receiving client.

## **2.2 Architecture models**

The goal of this thesis is to contribute to sharing files in a privacy-friendly manner. File sharing is done through one of two architectures: the client-server model and the peer-to-peer model. We investigate both to decide on which architecture fits our goals the best.

We assume peer-to-peer will be the better option as the client-server model will always rely on a trusted third party ('TTP'). However, when hosting that TTP privately in ownership of one's self, the dynamic changes entirely as opposed to trusting Google to be that TTP. Therefore, we also look into the self-hosted model, which will typically be a self-hosted variant of the client-server or peer-to-peer model.

### **Client-Server model.**

The client-server model works as the name suggests, there is a server (somewhere on the internet) and a client. For this thesis we leave server details out of the equation. So, a complete distributed server park with load balancing will be seen simply as a server with which the client can communicate with, as further details about the server would not contribute useful clarity to this thesis.

The client can communicate with the server and does all its business directly with that server. There is no intervention of other parties or services. In essence: one node dealing directly with one other node. Requesting a file in this scenario is simple: the client knows the location of the server and requests a file, the server responds with that file.

### **Peer-to-peer model.**

The peer-to-peer model works differently from the client-server model as it is distributed. There is no central server to communicate with, but instead a lot of other, similar to itself, nodes. Those nodes are both client and server concurrently for other nodes. Requesting a file is more complex in the peer-to-peer scenario. There is no known

server, therefore there needs to be some sort of distribution of a peer list. More information on this in section 4.1. After that, the correct node needs to be found and contacted for the file.

### **Self-hosted model.**

The self-hosted model means that a user sets up his own solution, for example a NAS (network attached storage), which is in essence a disk that is accessible over the network. So, users create their own server and clients and they remain in their control, thus eliminating the need for a trusted third party. Self-hosted approaches will usually adhere to the client-server model, such as a NAS, but it is also possible to self-host a distributed system in the network by using for example AFS<sup>3</sup>.

For our protocol we are most interested in the peer-to-peer and self-hosted models as they both eliminate the need of a trusted third party and offer complete control of privacy and security to the users themselves.

## **2.3 Existing solutions/products**

This thesis originates from the fact that no service currently in existence offers privacy-friendly file sharing. To verify this, an overview is made to summarize what existing solutions offer, how they operate and which architecture they use. Peer-to-peer based solutions are likely to offer the most usable information for this thesis and therefore extra effort is invested into finding peer-to-peer based solutions.

In an effort to keep the main body of this thesis concise and readable, the comparison table of all researched solutions can be found in appendix I. This section will contain a summary of that table, containing the most valuable lessons to be learned per architecture type.

### *Client-server model*

Within the client-server model, a lot of offerings can be found from a wide variety of companies. From smaller companies to companies as large as the first trillion-dollar company Apple [9] with their iCloud storage. These cloud storage solutions offer easy to use, well designed and cheap storage for every consumer. Often starting in a free tier for a small amount of storage and paid tiers for more storage. The consumers data is backed up by distributed datacenters that lay far away from the financial reach of a consumer [26]. Their data is always reachable from anywhere and from any device, making it a convenient solution.

However, in the current environment where data- and privacy breaches are getting ever more common [34], and where consumers are starting to distrust large non-transparent companies [15], cloud solutions are not a good starting point for attempting to achieve anonymous file sharing. Therefore, we quickly leap to peer-to-peer models. For more information about client-server providers, see appendix I.

---

<sup>3</sup> See the paper Analyzing the Tahoe-LAFS filesystem for privacy friendly replication and file sharing by “de Bruin” for an explanation of AFS (Andrew File System)

### *Peer-to-peer model*

Unlike the client server model, where many polished services are offered, the peer-to-peer model is rather underrepresented in ready-to-use solutions. A good example of a peer-to-peer implementation in a consumer-ready product is Resilio Sync. Resilio Sync uses peer-to-peer technology based on the BitTorrent protocol to allow users to share and synchronize data over multiple nodes without the need of a TTP whatsoever. They do offer TTPs for some tasks to make the program more useable for novice users but do not require or force the usage of those servers at all. Resilio Sync will be used as a starting point in this thesis and will be investigated in depth in section 4. One thing Resilio Sync does not provide is receiver anonymity, which we will look at in section 5.

### *Self-hosted model*

Within our research we found various solid solutions for file sharing such as OwnCloud/NextCloud and a network attached storage device or NAS such as a Synology. They can provide anything that a cloud storage solution can offer, but they cannot offer receiver anonymity. A self-hosted solution is more complex because users have to setup and maintain their implemented solution which, costs money and expertise which we cannot assume that an average consumer has. Another point to mention is that whoever sets the system up has full control over the system, so for other users it is still a TTP.

## **2.4 Privacy**

When it comes to privacy, we have to define what privacy means and what parts of it we want to protect with our protocol. Privacy itself is a vague term which is open to interpretation. For one person privacy might mean something different than for another.

Defining privacy and why it is a right has been difficult since at least 1890 when Warren et al. [32] wrote an article called “right to privacy” in the Harvard Law Review.

When looking at privacy in file sharing specifically, we have to consider the following objects/actors: sender, receiver and files (content). The content we want to keep private is the file, only the sender and receiver should know its contents. This can be achieved by encrypting the file with a key that only sender and receiver have. This is called confidentiality.

If we take it one step further, we might also want to protect the fact that the receiver has received a file, this is called receiver anonymity.

### **Anonymity.**

For anonymity we face a more difficult problem than confidentiality. As this is something the users cannot handle themselves by adding cryptography. There needs to be a system-level implementation that solves this problem for the users. When working within a peer-to-peer network, it is difficult to not disclose any personal information, in particular the users’ IP address.



Pfitzmann & Waidner 1987 [24] define three properties in anonymous communications: 1) sender anonymity, 2) receiver anonymity, 3) unlikability between sender and receiver. Sender anonymity and receiver anonymity mean that the identity of the sender or receiver is hidden whilst the other might not be. Unlikability between sender and receiver mean that although it is known that sender and receiver are communicating with some protocol, it is not known that they are communicating with each other.

Reiter & Rubin 1998 [25] add a factor to anonymity by specifying that there exists a degree of anonymity. They define a spectrum that ranges from “absolute privacy” to “provable exposed”. Between these two ends of the range there are also: “beyond suspicion”, “probable innocence”, “possible innocence” and “exposed”. It is hard to define which degree of anonymity is sufficient because it heavily depends on what the goal is. To evade prosecution for illegal downloading, “possible innocence” might be sufficient as the court cannot prove it beyond reasonable doubt. However, to hide a mistress from a wife, absolute privacy might be required. When looking into solutions for adding anonymity to file sharing, we will investigate which degree is achievable.

### 3 Goal of the system

The goal of this thesis is to contribute to a privacy-friendly way of sharing files. To make this goal and the “how to achieve” more precise, we define the following requirements/goals:

#### 1. Preserving anonymity of the receiver/downloader.

The primary goal is to provide anonymity for the downloader/receiver of the content. Receiver anonymity means that the receiver cannot be linked to any sort of content downloading at all. This might be too difficult to achieve, therefore we will first attempt to achieve unlikability between sender and receiver.

#### 2. Low need of computational resources.

The system should consume as little extra computational resources as possible, there should be a proper balance between anonymity and overhead. If possible, it would be nice to have an adjustable system where the degree of anonymity can be chosen dynamically. Too much overhead may prevent widespread adoption and drive end-users to other, more efficient, systems.

#### 3. Easy to use.

The system should be easy-to use for end users. Meaning the system should take care of the complex matters. Ease of use is important for this system as a lack of it may obstruct user adoption in favor for other systems such as Tor<sup>4</sup>.

#### 4. Free to use.

The system should be completely free to use or it should have a self-sustaining economical model. No initial investment or period-based subscription model should be necessary. This does also mean that the system needs a mechanism to protect against abusers that may, for example, attempt to (D)DOS the system.

#### 5. Anonymity before trusted third parties.

Trusted third parties may be used for this system, but when contacting them, the end-user should already be anonymized. The TTP should not be trusted for the anonymizing itself, but using a TTP to provide other tasks should be possible to keep flexibility in the functionality that can be offered.

---

<sup>4</sup> Tor stands for The Onion Router and makes anonymous communication possible by sending requests over a volunteered relay network

**6. Other privacy principles should be possible.**

To protect the defined scope for this thesis, the system should provide anonymity. Other privacy principles, such as confidentiality and integrity of data, should be possible but are an added bonus, not the focus of the system.

**7. Provide anonymity against a non-global adversary**

The system should provide anonymity against a non-global adversary. This means an adversary can take part in the system just a regular node would. It can act as multiple nodes and have all information a normal node has. This non-global adversary is similar to other anonymity providing systems such as Tor [10].

## 4 BitTorrent & Resilio Sync

As we saw in section 2.3 Resilio Sync<sup>5</sup> appears to be a good starting point for our system. It is one of few existing ready to use and peer-to-peer based systems for file sharing, synchronization and backup<sup>6</sup>. In this section we will take a closer look at Resilio Sync and the BitTorrent protocol to discover what privacy they offer for their users and where improvements can be made with regards to anonymous file sharing.

### Privacy according to documentation.

We will first investigate what Resilio Sync communicates to their users through legal documents. This will give us a good starting point to figure out what privacy Resilio Sync offers.

When installing Resilio Sync the user must agree to the following three documents: privacy policy<sup>7</sup>, terms of Use<sup>8</sup> and the end user license agreement (EULA)<sup>9</sup>.

When looking through all of these documents we find that they are readable with clear language that most users could understand.

#### *Privacy Policy*

The privacy policy mainly focusses on the website of Resilio, on what they store, why they store it and how they might use it. Although the privacy policy that Resilio has is reasonably standard, one might expect Resilio to respect privacy better since that is one of their key selling points. Some examples of what Resilio collects from site visitors:

- Collecting: “Internet Protocol address, browser type, browser language, the date and time of your query and one or more cookies”
- “count, track, and aggregate the visitor's activity into our analysis of general traffic flows”
- “collect information about your computer or mobile device, such as your device model, browser type, or sensors in your device like the accelerometer”

The privacy policy does not say much about their client software other than:

- “We do not track which files you transfer with the Resilio Client”
- “We also aggregate some data from the Resilio Client regarding total traffic flows and content delivery performance as well as other data collected in the use of our products or services”
- The client also uses cookies and third-party cookies

#### *Terms of Use & EULA*

The EULA contains no text about privacy related matters at all.

The Terms of Use also do not state anything about privacy other than that they have no responsibility or liability regarding security and privacy.

---

<sup>5</sup> <https://www.resilio.com/>

<sup>6</sup> Resilio Sync does not offer backup directly, but a modified one-way file syncing can be used.

<sup>7</sup> <https://www.resilio.com/legal/privacy/>

<sup>8</sup> <https://www.resilio.com/legal/terms-of-use/>

<sup>9</sup> <https://www.resilio.com/legal/eula/>

Unfortunately, the legal documents do not tell anything about the privacy guarantees their products provide, but only how their website deals with privacy. This does not tell us if, when sharing a file, the receiver is anonymous or not. They, logically, focus on how they as the software provider offer privacy.

To discover what privacy the product itself and the protocol Resilio based their solution upon offers, we will dive into the technological side of Resilio Sync in the next section.

### **Privacy in technology**

Resilio Sync was formally called BitTorrent Sync [4] which gives a good hint where to look for which protocol it's based on, namely the BitTorrent protocol. Before diving into what BitTorrent Sync/Resilio Sync<sup>10</sup> is and how it works we first look into the underlying protocol on which it is based to get a complete understanding of how it works, which additions are made to the original protocol and what privacy it offers.

#### **4.1 BitTorrent Protocol**

BitTorrent is a very popular distributed file sharing protocol that utilizes the peer-to-peer model. With the help of a centralized server, which users use to find each other, users can download files from each other. To make this downloading efficient, files are split into smaller pieces that users can request from multiple other users concurrently.

Unfortunately, the BitTorrent protocol is often used to spread copyrighted content and therefore several "torrent sites", such as thepiratebay.org, are under constant scrutiny and even blocked in some countries by ISPs [27].

We are not interested in any controversies regarding the usage of the BitTorrent protocol but solely in how the protocol works and if it can help us with our quest for anonymous file sharing. We start with a short summary of how the protocol works, followed by an in-depth description.

#### **BitTorrent in short.**

BitTorrent is a rather simple protocol. We will explain BitTorrent by using a scenario. A user wants to share a file, located on their local machine over the internet in a distributed fashion, instead of using a centralized cloud-based solution. To achieve this, the user makes its file available for download from their machine by installing the BitTorrent client, which listens to a specific port on the network for requests. The BitTorrent client also assures that any firewalls are configured correctly. By activating the BitTorrent client, the computer becomes a so-called peer in the network.

Users that wish to download or receive files also need to run a BitTorrent client, to become a peer as well. After becoming a peer, the receiver only needs the sender's IP address and correct port to send a request to. The first problem that arises is the distribution of that IP address and port. The downloading peer does not know these

---

<sup>10</sup> When this paper references to BitTorrent Sync or Resilio Sync the same protocol is meant. To prevent confusion Resilio Sync will be used predominantly

beforehand and therefore cannot start the download. To solve this problem, BitTorrent creates a known central server (called a tracker) that helps with distributing that information. The sending user announces their IP address and correct port to this server and the receiving user requests the location where to send its download request to from this server. So, these known central servers' only job are to keep track of information about the sending peer and to answer requests for that information. It does not store or distribute files to be shared itself.

Of course, the file is shared with the entire internet, not just one user, so we need to be able to handle many requests at once. The sending users' internet bandwidth is limited and can only serve a limited set of peers at the same time. To mitigate this problem BitTorrent tells other users who have already downloaded the file to also become a sending user. To do this, they also announce their IP address and correct port to the known central server which allows receiving peers to get a list of download locations which they can probe for download speed and availability. Based on these probes they can choose a sending user with the fastest download speed to start the actual download from.

Next thing to notice is that user availability becomes a problem. A user might go offline whilst another user is in the process of downloading a file from them, meaning that the downloading user must start over again, from scratch. Since large file downloads can take up quite some time, this situation is not ideal. Therefore, BitTorrent chops the file into several smaller pieces that can be downloaded individually from several different users. Users do this by keeping a list of which pieces they have and need, and requesting the ones they need. To optimize speed even further, users can request multiple pieces to several different users concurrently

We do, however, notice a problem with the aforementioned solution. If, for example, the first node is uploading a 1000GB file to another node, users have to wait until that second user is done with downloading before they can send requests to that user as well. This is an undesirable performance reduction. To solve this BitTorrent allows users to also download from incomplete users. This does, however, result in a new problem: how does a user know which other users have which pieces? To solve this all users announce their local pieces list to other users who in turn save that locally. A user can then check which pieces of the file it needs and whom to send the request to. This means that the difference between sending and receiving peers disappears as peers can act as both at the same time.

We now have a performance optimized downloading network where each user contributes to the network by uploading. This, in short, is how BitTorrent works. In the remainder of this section we go into more detail how each component works.

### **Detailed look into BitTorrent**

The first problem we came across in our summary is called "peer discovery", but that was based on the assumption that the nodes already know each other and that they want to download content from each other. BitTorrent, however, does not offer any type of content searching or content hosting whatsoever, content discovery is instead outsourced to specific torrenting sites. This lack of content localization allows the BitTorrent protocol to focus solely on distributing files and doing so efficiently [18]. To

aid this workflow, BitTorrent introduces .torrent files. These files can be downloaded from a torrent site and contain all the information needed for a peer to initiate the download, thus to know from which server to request a list of IP addresses and ports from. We will not investigate content distribution sites as they fall outside of the BitTorrent protocol. The starting point of any BitTorrent protocol will be either downloading or creating a .torrent file.

### **.torrent or metainfo file**

The .torrent file or metainfo file holds information in bencode (bencode is a simple and flexible encoding standard used by BitTorrent) about the location of the tracker and information about the files that can be downloaded. It is meant to get information to peers who want to join the BitTorrent session in an efficient manner.

The .torrent file contains the following bencoded static dictionaries, which do not change over the life of the torrent session [33]:

- **Announce** - URL of the tracker (this can also be a list of multiple trackers);
- **Info** – maps to a dictionary with the following keys:
  - **Name** – string with the suggested name of the file;
  - **Piece\_length** – number of bytes of the pieces that the file is split into;
  - **Length** – size of the file (only present when torrent contains one file);
  - **Files** – a list with one dictionary per file (only present when torrent contains multiple files) containing the following keys:
    - **Length**;
    - **Path**;
- **Pieces** – a list of SHA1 hashes for each piece;

When initiating the download, a peer needs to allocate space for the file or files because the files will be downloaded in a random order [28]. After downloading the torrent file, the peer can continue with peer discovery which is explained later in this section.

### **Creating a .torrent file.**

Creating a .torrent file is very easy, most BitTorrent clients have built in functionality to handle this process. The user chooses one or more files to share and they select one or more trackers to use for that torrent. The client then creates a torrent file, in bencode, containing the keys mentioned in the previous section. All are trivial to fill, except for piece\_length and pieces.

Piece\_length is not explicitly defined in the BitTorrent protocol, but rather a value for the client to decide. Larger pieces mean less overhead on making TCP connections and thus less resources are wasted, but when there are too few pieces because of this, the peer-to-peer protocol runs less efficiently because there needs to be a certain amount of pieces to get an optimal sharing climate between all the peers. After years of experimenting, the optimal piece amount seems to be between 1200 and 2200 [13].

The pieces key can be filled by splitting the file(s) into pieces with the correct piece\_length and then hashing each piece and concatenating the values to form a long string. This means that smaller piece sizes equal larger .torrent files [18].

After the creation of the torrent file, it needs to be announced to at least one tracker (more are possible). After announcing the torrent to the tracker(s) the user should stay online long enough to seed the files to other peers. As soon as there is one other peer that downloaded the complete file and start seeding, the initial node can leave the session [18].

### **Peer discovery.**

Now that we know how to create a torrent session and how to get the information necessary to join one, we are moving on to peer discovery. BitTorrent provides a couple of methods for peer discovery. We will briefly describe them, followed by a more in depth look.

The first is called a tracker which is a central server that holds information about the peers in the torrent session and answers requests from peers that desire that information.

Another method is called PEX (peer exchange). During communication regarding a torrent, peers also exchange peer lists with each other. This is done after a peer has joined a torrent session and can therefore not be used to initiate a torrent download.

To discover local peers in the LAN instead of over the internet, LDP (local peer discovery) is proposed. It works by sending multicast packets over LAN. As soon as another BitTorrent client receives one, it checks if they have overlapping torrent sessions and makes a connection if so.

The final method is called DHT (Distributed Hash Table), it can be seen as a look-up table for peers. This hash table is distributed and maintained by active peers.

### *Trackers.*

Trackers are centralized servers (not distributed peers) used by the BitTorrent protocol for peer discovery and therefore make BitTorrent a hybrid protocol [33].

After obtaining the .torrent file from outside the BitTorrent protocol a HTTP GET request is sent by the peer to the tracker containing the following keys (again in bencode) that are obtained from the torrent file or information about the node itself [28]:

- **Info\_hash** – identifier for which torrent session is subject;
- **Peer\_id** – random ID of the client;
- **IP** – optional value giving the IP or DNS name;
- **Port** – port number on which the peer is listening;
- **Uploaded** – total data uploaded so far;
- **Downloaded** – total data downloaded so far;
- **Left** – number of bytes left to download;
- **Event** – optional value containing status (started, completed, stopped or empty).

The tracker responds to this HTTP GET request with bencoded dictionaries containing two keys: 'interval' and 'peers'. The 'interval' key is the number of seconds that the peer should wait for their next request [28]. The 'peers' key maps to a list of dictionaries of peers that each contain the following keys, peer\_id, IP address and port [28].



A tracker can handle a lot of torrent sessions concurrently because the required bandwidth is very low. A peer, usually with a 30 second interval, sends a GET request which takes a short time to answer the rest of the download process is done without the tracker.

To reduce overhead of the trackers TCP requests, a proposal has been made and accepted through BEP's (BitTorrent Enhancement Proposals) for an alternative in the form of an UDP based protocol that reduces overhead by 50% [30].

Unfortunately, from a privacy perspective, the tracker knows a lot of information about the peers that prevent anonymous file downloading. Also other peers get to know the IP address and ports of all other peers downloading the file(s).

Trackers in the original BitTorrent protocol are publicly available for everyone, but an addition to the protocol has been made to create private trackers which add value by filtering spam and viruses, guarantee better privacy by not publicly listing peers their IP address and offer faster downloads by requiring peers to also upload to a certain ratio. For these private tracker's users need an account to authenticate with when sending requests to the tracker [[7]].

#### *DHT (Distributed Hash Tables).*

An alternative for using trackers has been proposed and accepted through a BEP which allows trackerless torrents that utilize a DHT. Peers use the BitTorrent info\_hash key to store their IP address in the DHT.

A DHT needs to replace the functionality of a tracker, which is to respond put and get requests. The DHT interface offers exactly that. With a DHT the "put" request stores the IP address with the corresponding IP address and the "get" request looks up an info\_hash and return a list of IP addresses that are stored under it. The main benefit a DHT in the BitTorrent protocol offers is that it eliminates the need for a TTP and puts all aspects of the protocol in a peer-to-peer setting [33].

#### *PEX (Peer exchange).*

PEX cannot be used by a peer to join a BitTorrent session. It will need to find peers either through a tracker or a DHT before PEX can start operating. This is because PEX only send messages between peers. It allows connected peers to exchange information about other connected peers in the session directly without polling the tracker or the DHT. PEX works as follows: a peer sends a request to one of its connected peers to request information. That other peer responds with a list of peers they are connected to and disconnected from since the last request. This allows a session to continue operating when the tracker goes offline and reduces load on the tracker [33].

#### *LDP (Local Discovery Protocol).*

Finally, there is a peer discovery method that operates only over the LAN. LDP helps with finding local peers within the LAN which is beneficial because they are likely to have fast up and download speeds. BitTorrent calls this technique LSD (Local Service Protocol). It works by sending an "LSD announce" over the LAN to each interface on which it is listening. Upon receiving an LSD announce, the peer will verify if it is participating in the same torrent session by looking at the info\_hash sent with the announce. If this is the case an attempt is made to initiate a connection [33].

### Peer communication.

After the peer has obtained the peer list from the tracker, or through another peer discovery method, further communication is not required between the peer and peer discovery methods, it does still take place on a set interval to get an updated list.

The peer now must contact other peers on the list to obtain the actual file. The communication between peers is done following the specification of the BitTorrent Peer Protocol which operates over TCP [28]. Peer connections are symmetrical, meaning messages can flow in both directions and the messages look the same in both directions [28].

For ease of explanation we will have a receiving/receiver peer and a sending/sender peer, of course in the actual protocol each peer fulfills both rolls.

The protocol starts with a handshake to establish a connection. Besides initializing a connection between the two peers this handshake also establishes which torrent is subject of the connection. The handshake is sent by the receiver to a subset of peers from the peer list that the tracker provided and contains the following:

- **Pstrelen** (in v1.0 of the protocol this is 19);
- **pstr** (in v1.0 of the protocol this is “BitTorrent protocol”);
- **8 reserved bytes** (currently all zeros);
- **info\_hash** (unique hash-based identifier to indicate which torrent is subject);
- **peer\_id** (unique id of the peer).

As soon as the recipient sees info\_hash, it must respond. If it is not currently serving that info\_hash, it must drop the connection [28]. After a successful handshake a never-ending stream of length-prefixed messages may follow.

The first message that is sent by the sender is a ‘bitfield’ message which contains a payload that indicates which pieces the sender has. The payload consists of bytes where the bits in the byte represent the index of the piece, so the first byte indicates index 0-7 [28]. If the sender has the piece, the bit is set to 1. If not, then to 0. The receiving peer stores this bitfield locally to know which pieces it can request from this specific sender and this information is also used to select the rarest piece first which will be explained later in this section.

A connection starts by default in the ‘not interested’ and ‘choked state’. To better understand the protocol, we first look at what these states mean. ‘Not interested’ and ‘interested’ are states that the receiver announces towards a sender to indicate that the sender has pieces that the receiver needs. The ‘choked’ and ‘unchoked’ states are states that the sender announces to all receiving peers. As mentioned, a connection starts in the ‘not interested’ and ‘choked state’ meaning that before any actual request messages can be sent or answered, the receiver peer must first indicate that its ‘interested’. The sender can then determine from its peer list which are ‘interested’ and unchoke those connections. How and which peers to unchoke is decided according to the unchoke algorithm which will be treated later in this section.

The receiving peer now has a list with connected peers that are choked, it selects which one has pieces that itself does not have and send an ‘interested’ message to that sender. The ‘interested’, ‘not interested’, ‘choke’ and ‘unchoke’ message have no payload. The receiver then must wait until a sender sends the ‘unchoke’ message before it

can start sending 'request' messages. These 'request' messages contain the following keys; index, begin and length. Index indicates which piece of the file is requested, begin and length specify which block within that piece is requested. The sender replies to that 'request' with a 'piece' message that contains index, begin and the piece itself. After completing a request, the receiving peer has a new piece of the file, it checks the integrity by hashing that piece and comparing it to the pieces dictionary that was acquired from the .torrent file. The receiving peer then needs to update all connected peers that it has a new piece available so they can indicate if they are interested in that piece. This is done by sending 'have' messages where the payload is the index of the piece it just downloaded and verified.

When nearing the completion of the download, 'endgame mode' is started to assure that the final pieces come in quickly. This is done by sending piece requests for every missing piece to every connected peer. To keep this efficient a 'cancel' message, with the same payload as the 'request' message, is sent as soon as the piece is received. After the last piece is received and the transfer is complete, the receiving peer can decide to stay as a seeder or leave the torrent session.

A distinction can be made between two types of sending peers: 1) peers who are currently downloading the file ("leechers") but also upload the pieces they already have and 2) peers who have the full file downloaded and upload it (seeders).

#### **Piece selection algorithm.**

BitTorrent uses a "rarest piece first algorithm" to assure a higher availability of all pieces. For example, if one seeder has a piece that no else has and he ends his torrent session it would mean that all other peers can never complete their download. Whilst if all peers investigate what the rarest piece is and request that whilst the sending peer is still in the torrent session, that leaving peer is no longer crucial for the overall download process.

There is, however, an exception case, when a new peer joins the torrent session. It needs to have a first piece as soon as possible to be able to unchoke connections, therefore it will download three pieces at random before starting to look for the rarest piece.

This algorithm works rather simple because a receiving peer has a list of all other connected peers and which pieces they have. To find the rarest piece a peer counts on how many peers their pieces list it appears. Then select the piece with the least occurrences.

#### **Choke algorithm.**

As mentioned before, a vital part of the BitTorrent protocol is the unchoking algorithm as it is responsible for a sender to decide which downloaders can request pieces from it. In BitTorrent there is a default number of peers to unchoke at one time, namely four. The problem that the choke algorithm must solve is which four peers to unchoke. This is important as it affects download and upload speeds and how fair a peer is. Peers that only download and never upload are unfair and should be left choked.

Decision on which peer to unchoke are strictly made on download speed [6]. Calculating download speed is done by taking a 20-second average. To avoid wasted

resources because of rapid choking and unchoking those actions are only done every 10 seconds which is enough for TCP to ramp up a new transfer to full capacity [6].

This does, however, leave a lot of connections untested for their upload speed. To solve this there is an optimistic unchoke that is done every third rechoke period. The optimistic unchoke is done by choosing a random peer that is not amongst the four selected peers.

The last problem to overcome is “snubbing”, which means that all connections which the peer previously downloaded from are choked. In such a case it will continue to get slow download rates until the optimistic unchoke finds a faster peer. The mitigation for this problem: when no piece has been received from a peer for over one minute to stop uploading to that peer. This allows for more optimistic unchokes that result in a faster recovery of the download speed [6].

### **BitTorrent Privacy Analysis.**

Now that we have a good understanding of how the BitTorrent Protocol works we can start investigating privacy within the protocol. First, we will look into what identifiers the client has. When looking into all communications between peers and trackers we find the following identifying objects:

- Peer\_id;
  - Is randomized for each torrent session, so there is no likability between torrent sessions;
- IP address;
  - Is a unique identifying property that is linkable over all torrent sessions;
- Port;
  - Is the same for a large group of peers so a peer is anonymous within the anonymity set of peers that use the same port.

So, the IP address of a peer is what we will eventually need to protect to assure more privacy within this protocol. It is sent to the tracker and to all other peers in the torrent session. The following information can be learned about a peer: what they download, when they download (at what times of the day is the PC on), how fast they download and if they are fair (up- and download ratio).

The BitTorrent protocol offers no end-to-end encryption, no transit encryption other than standard TCP and no encryption for stored data.

There is not authentication mechanism and IP addresses are the basis of identifying peers.

## 4.2 Resilio Sync protocol

Resilio Sync<sup>11</sup> is a modified version of the BitTorrent protocol. The focus of BitTorrent lays primarily on sharing files one-to-many in a public setting. Whereas the Resilio Sync protocol focusses more on file synchronization in a private setting. The private approach is done by adding authentication and encryption to the sharing process. Resilio Sync is not open sourced and the operation of the technology is not documented by the developer [26], therefore information is hard to find. We rely on information provided by Resilio Sync's own documentation and scientific papers. Our focus will be to research the modifications made to the BitTorrent protocol.

### **Resilio Sync in short.**

Resilio Sync is meant as a private peer-to-peer platform with the ability to synchronize and share files with other peers. Making it private is done by adding cryptography over folders and with an authentication mechanism to assure only authorized peers can download the content.

In the BitTorrent protocol, finding torrents to download is done through torrenting sites that host .torrent files. This setup would not work for Resilio as it has a different goal, namely private file synchronization. Therefore it needs a new sharing mechanism to replace the .torrent files used by BitTorrent. Resilio resolved this by sharing a file by sharing a link (URL) that contains all the information a client needs to initiate the synchronization session, apart from the peer list. Peer discovery is done similarly to BitTorrent, using a tracker, discovery over LAN or directly setting the IP address manually.

To further assure private file sharing, Resilio introduces a key system in which each "torrent session" gets its own set of keys. Sharing those keys means sharing access to the content. A temporary key is included in the share link that enables the receiver to connect to the file owner and request access to a file. After the sender approves that request, it generates a certificate for the receiver, signs it and makes sure it has the correct permissions (from "no access" to "owner")<sup>12</sup>. Links can also be published on the web in a .torrent file fashion with the approval process turned off, to allow a more public type of sharing.

Resilio offers a permission system where peers can be given for example read-only or read-write permissions. These permissions also extend into encryption of the content that is being shared. For example, a read-only permission for encryption allows a peer to download the file and decrypt it. Another example is the copy-only key that allows a peer to copy the data but not decrypt it thus functioning as a backup.

Resilio Sync is more focused on file synchronization than only file sharing. It attempts to replace services like Dropbox and Google Drive. File sharing is still possible by synchronizing a file with a read-only key.

---

<sup>11</sup> Previously called BitTorrent Sync

<sup>12</sup> Resilio Sync Help Center - <https://help.resilio.com/hc/en-us>

**Detailed look into Resilio Sync.**

One major difference between Resilio Sync and BitTorrent is authentication. Resilio Sync uses authentication methods to verify if someone is entitled to download certain files. To accommodate this, an identity is made for each peer to be able to verify if that identity is entitled to join the session.

**Identity creation**

When installing, Resilio Sync asks the user to give an identity name. That name is used for creating a digital certificate and a random fingerprint<sup>13</sup>. This certificate is used for requesting access to files from other peers and to grant access to files to other peers. It is the basis of the authentication system within Resilio Sync. For free users one identity per device is required. However, for “pro users” (paying users) the option exists to use an identity over multiple devices. This has the benefit that when creating a file on one device, it is automatically available on all other devices with the same identity, preventing the need to share every file with every device. The identity can be transferred using the so-called M-key. Free users have to share folders across multiple identities manually for each folder they want to share<sup>13</sup>.

**Keys.**

After the initial setup process, a user has the client installed with an identity and can start with sharing/syncing folders. For each folder that a user creates, Resilio generates a set of folder-specific keys using ED25519 and SHA3 cryptographic protocols<sup>13</sup>. A prepended letter indicates the type of key.

For a standard folder:

- A, Read & Write key (RW), master secret;
- B, Read Only key (RO), derived from A.

For an encrypted folder:

- D, Read & Write key with capability to seed to encrypted nodes (RWE);
- E, Read Only key with capability to decrypt data (ROD), derived from D;
- F, Encrypted key (EK), can receive, store and seed data but not decrypt it, derived from E or F;

Next to the aforementioned keys, a share ID based on the RO key is generated. A traffic encryption key is generated per session and is also based on the RO key<sup>13</sup>. These keys are necessary when sharing files with other peers as we will see in the remainder of this section.

**Sharing.**

As mentioned before, sharing is not done through a .torrent file but instead a special link is created. This link can be shared with the receiver of the files directly, this

---

<sup>13</sup> Resilio Sync Help Center - <https://help.resilio.com/hc/en-us>

distribution is done outside of the Resilio Sync protocol. Links are normal URL's that can be opened in the browser, in which case the browser sends it to the client. Or directly pasted into the client. Example of a link followed by an explanation:

*https://link.resilio.com/#f=Example%20Folder&sz=45E4&t=1&s=RUXP62GFKHCPHQNMRL2I7QFEKSHBH2N5&i=CPZIW2F2QIIS5EK6Y3QKVXIQ3JH5GM2K&e=1561992430&v=2.6*

The link starts with the domain name to allow the Resilio site to open the client for added user comfort. It is followed by *f*, which represents the folder name, *sz* is the approximate size in exponential format, *s* is the share ID (similar to the BitTorrent info\_hash) and *i* is the temporary key that the receiver uses to connect to the sender. *e* is the expiration time in Unix time format and finally *v* for the version of the client used to send the file.

### **Peer discovery.**

After the setup phase, where an identity is created and after a folder is shared through a link, it is time for peer discovery. Resilio sync utilizes three methods for this<sup>14</sup>, two of which are very similar to the BitTorrent protocol.

- **Tracker server** (modified version of the tracker from the BitTorrent protocol);
- **LAN Discovery** (similar to LDP that BitTorrent uses);
- **Predefined hosts** (new technique that would not work in BitTorrent).

#### *Tracker server.*

This is an optional service offered by Resilio Sync, which is turned on by default, it works very similar to the trackers in the BitTorrent protocol. When the client query's the tracker, which is a TTP provided by Resilio, it sends the Share Id as well as the internal IP address and port. The tracker determines the external IP address and port automatically. The tracker responds with the information of other peers and their IP addresses and ports. The peer can now try to connect directly with the other peers, first a LAN connection is attempted otherwise an over the internet connection is attempted<sup>14</sup>.

#### *LAN Discovery.*

LAN discovery is not optional, every Resilio client subscribes to and send multipacket over port 3838, within these multipackets the Share ID, internal IP address and port are sent. This enables a client to listen to the multipackets and as soon as it sees one with a Share ID it is looking for, a connection is made with the sending peer. Peers send multipackets as soon as their client starts and whenever a network change is detected<sup>14</sup>.

#### *Predefined hosts.*

Predefined hosts is an optional service by Resilio Sync, which by default is turned off. It allows users to give the IP and port of another peer instance manually. Thus overriding the need of the tracker server<sup>14</sup>.

---

<sup>14</sup> Resilio Sync Help Center - <https://help.resilio.com/hc/en-us>

**Peer connection.**

In previous sections we looked at how sharing a file through a link and peer discovery works. The next step is to initiate communication between peers to transfer the file.

The receiving peer starts the protocol by sending a “request for folder access” to the sending peer with the temporary key from the share link. That request includes its local generated public key and username. The sender peer must approve the request (if that option is enabled) based on the identity of the receiving peer. After approval, the receiving peers receive a copy of the required file encrypted with their public key [26]. The sender then adds the receivers identity to its ACL (access control list) and signs it with its own certificate granting the correct permission.

**Relay server**

In the case that there are connection issues between two peers, for example firewall, proxy or NAT issues, Resilio Sync offers a Relay server that reroutes all traffic from one peer to the other through their relay server. Because all data that is in transit is encrypted, the relay server cannot see the content at all according to Resilio<sup>15</sup>. The peer that has the data sends it directly to the relay server, the relay server then sends that data to the receiving peer and therefore eliminating direct connection between peers. This does make it appear more like a client-server model; however it is still considered peer-to-peer. Resilio claims that after successful data transfer the data is immediately deleted from the relay server<sup>15</sup>.

**Encryption.**

When looking into the encryption that Resilio Sync uses we are pleasantly surprised with their implementation, especially because the BitTorrent protocol that it is based on does not use any encryption whatsoever.

Resilio Sync uses the following encryption for their service:

- Data in transit is AES-128 encrypted using the traffic encryption key<sup>15</sup>;
- Data stored remotely is AES-128 encrypted using a storage key<sup>15</sup>;
- X.509 certificates are used for mutual authentication<sup>15</sup>.

So, Resilio uses encryption to share/synchronize data. In the case of a normal folder; the data is encrypted with the traffic encryption key and sent over an SSL connection directly to the other peer (or in some cases the relay server). In the case of an encrypted folder; the data is encrypted with the storage key, then it is encrypted again with the traffic encryption key and sent over an SSL connection to the other peer (or relay server).

**Privacy in the Resilio Sync protocol**

When looking at the privacy offered by Resilio Sync we can divide the privacy into two groups. Peer-to-resilio privacy and peer-to-peer privacy.

---

<sup>15</sup> Resilio Sync Help Center - <https://help.resilio.com/hc/en-us>



*Peer-to-Resilio privacy*

There are several possible points of contact with the Resilio infrastructure<sup>16</sup>:

- Tracker Server;
  - The tracker server learns, internal and external IP addresses and ports of all peers of a shared folder;
- Relay Server;
  - Learns the source and destination IP addresses and ports of peers;
- Check for update;
  - Can be replaced with manual client updates;
- Link landing page;
  - Can be circumvented by pasting link directly into the client;
- License purchase server;
  - In a completely open sourced implantation this can be removed.

Important to note is that all of the above options can be turned off meaning Resilio will know nothing about a peer<sup>16</sup>.

*Peer-to-peer privacy*

Peers learn each other's IP address, port and client identity, which mean anonymity between peers does not exist within the Resilio Sync protocol.

**Table 1.** Design goals – Resilio sync vs BitTorrent

Design goal #	Resilio Sync	BitTorrent
1. Receiver anonymity	-	-
2. Low computational resources	-	+
3. Easy to use	+	+
4. Free to use	-	+
5. Anonymity before TTP	-	+
6. Other privacy possible	+	+
7. Anonymity against adversary	-	-

**Possible solutions.**

As we can see in table 1, both Resilio Sync and BitTorrent do not meet all of our design goals from section 3. Resilio Sync does not meet several of the design goals but BitTorrent is the better contender of the two as it only misses design goal 1 and 7. The core of design goal 1 and 7 is anonymity. So we need to add anonymity to BitTorrent to get a satisfactory result. To achieve anonymity a couple of solutions jump to mind, such as mix nets or crowds. When looking into existing literature we find an implementation that looks promising for crowds within the BitTorrent protocol called BitBlender by Bauer et al. [2]. We investigate BitBlender in the next section.

---

<sup>16</sup> Resilio Sync Help Center - <https://help.resilio.com/hc/en-us>

## 5 BitBlender by Bauer et al.

The BitBlender protocol by Bauer et al [2] is layered upon the BitTorrent protocol. We will look into applicability for Resilio Sync later in this section. BitBlender uses “crowds” as described by Reiter & Ruben [25] to add a certain level of anonymity. The design goals of BitBlender are the following: more lightweight in computational resources compared to other anonymous protocols, easy to use, work seamlessly with the existing BitTorrent protocol, offer plausible deniability and finally offer tunable anonymity [2]. To understand how BitBlender achieves anonymity, we first briefly look into crowds.

### Crowds.

Reiter & Ruben have proposed a system called “crowds” [25]. This system is based on the idea that when a person is in a crowd it is no longer identifiable as a single person. For web requests they propose that the person submits its request to another person in the crowd who then either sends it to the server or another random person in the crowd. When the request eventually reaches the server, it is sent by a random member of the crowd (not the initiator), so the server does not know who actually sent the request. Each member in the path of the request also does not know who the request came from, it might be the one that directly requested it to them, but that might also be just a forwarder.

### BitBlender in short.

BitBlender uses the crowd’s system to introduce anonymity to the BitTorrent protocol. Within BitTorrent there is already a crowd, but it consists of only active peers that are actively participating, so in this case both the receiving and sending peer know that the other peer is actively downloading and thus there exists no anonymity.

BitBlender’s solution to this is to add so called “relay peers” (inactive crowd members) who do not actively participate but only forward requests to other peers, again either to another relay peer or an active peer just like in the crowds system. In this case both the sender and receiver cannot identify if the request comes from or goes to a relay peer or an active peer. In figure 2 the relay peer represents one or more relays.

Fig. 1. - BitTorrent request

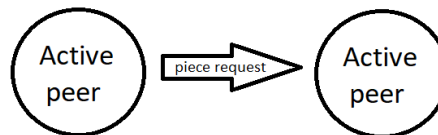


Fig. 2. - BitBlender request



This means unlinkability between sender and receiver is achieved as one cannot prove a peer to be actively down- and uploading or just relaying messages.

Also, the peer lists that trackers return now contains relay peers and therefore one cannot tell for sure that a peer from the list is actively downloading/ uploading so their design goal of plausible deniability is achieved. We will now take an in-depth look into BitBlender and at the end of this section we look for flaws and shortcomings.

### Detailed look into BitBlender.

BitBlender is based upon the BitTorrent protocol directly, so for now we ignore Resilio Sync, we will discuss that later. Our starting point is a fully functioning BitTorrent protocol that utilizes trackers for peer discovery. In essence all BitBlender has to do is add relay nodes to the set of all nodes, but their design goals make this a bit less trivial. Because normal peers that are unaware of BitBlender should still be able to function normally and the number of relay peers added to a session should be dynamically adjustable per session to have tunable anonymity. BitTorrent has two components to work with, the tracker and the peers. BitBlender cannot modify the existing group of peers in order to keep normal peers able to use the protocol, but they can modify the tracker and add a new type of peer.

### Tracker adjustments.

The tracker knows how many peers are in the session and it can be told a certain level of anonymity is desired. So, the tracker is the perfect place to attract relay nodes, or is it? If the tracker requests the relay nodes itself, then it also knows which nodes are relay nodes, which is not ideal. To overcome this problem BitBlender borrows the blender that Reiter & Ruben [25] defined in their crowds protocol. The tracker is expanded to hold responsibility for calculating how many relay nodes are required and to request them from a blender. Let  $N$  be the set of active peers and  $M$  be set of relay peers such that  $N \cap M = \emptyset$ . This request is a tuple of  $(n, t)$  where  $n = |M|$  and  $t$  is unique identifier for the tracker [2] (the announce URL).

### Blender design.

The blender holds the set of all relay peers called  $B$ . The blenders task is to make sure that  $n$  relay nodes join the torrent session. It could do this by simply instructing  $n$  nodes from the set  $B$  to join the session by sending a request to them. In that case the blender knows which nodes in the torrent session are relay nodes and therefore becomes a single point of failure.

To overcome this problem BitBlender has defined the following solution. When the blender receives the request from the tracker, it starts by calculating the join probability  $p$  where  $p = \frac{n}{|B|}$  [2]. It sends this  $p$  to every  $b_i \in B$ , each relay node then decides based on  $p$  if it will join or not (see relay nodes section on how).

For the design of the blender the paper uses a single centralized directory server for the sake of simplicity but do note that this means there is a single point of failure for availability [2]. Alternatives are to replicate the blenders database over the network and

use a consensus technique to run queries [2]. Another possible solution is that the blender might exist as a service via a distributed hash table, but that leaves it vulnerable to Eclipse attacks [2]. For adding relay peers, the exact implementation is not important and therefore a simple server is chosen as example.

### Relay nodes.

Upon receiving a request from the blender with  $p$  every  $b_i \in B$  chooses a pseudorandom number  $r \in \mathbb{R}$  where  $0 \leq r \leq 1$ . Then iff  $r \leq p$  that relay node joins the torrent session [2]. Resulting on average  $n$  relay peers joining the session without the blender knowing which ones.

When relay peers have joined the torrent session, they appear to function as normal peers and are not distinguishable from active peers. However, instead of participating in the protocol, they simply forward any requests to another peer. That other peer cannot tell if that request is from an active peer itself or a relay peer.

Relay peers also appear on trackers, indistinguishable from normal peers [2]. “An ad-hoc relay network is constructed in this manner, where the path lengths are probabilistically influenced by the concentration of relay peers to real peers in the torrent.” [2].

### Path length.

The path length is dependent on the amount of relay peers in relation to the total peer number. BitBlender assumes, for simplicity, that piece requests are sent to a random peer from the set of all peers. Let  $N$  be the set of all peers (relay and normal). Let  $M$  be the set of relay peers such that  $M \subseteq N$ . Let  $P$  be the set of normal peers such that  $P \subseteq N$ ,  $P \cap M = \emptyset$  and  $|P| + |M| = |N|$ . The path always starts in  $p_i \in P$  through 0 or more  $m_i \in M$  and ends in  $p_j \in P$ . The path length  $l$  is dependent on the ratio of  $|M|$  to  $|N|$ . This ratio is defined as  $r = |M| / |N|$  [2].

The formula for the path expected length is the following [2]:

$$E[l] = \frac{1}{1-r}$$

Some examples: if half of the total peers are relay peers, then we get a ratio of  $\frac{1}{2}$  and thus an expected path length of 2. If three quarters of the total peers are relay peers, we get ratio of  $\frac{3}{4}$  and thus an expected path length of 4 [2].

### Compatibility with the existing BitTorrent protocol.

One of the design goals of BitBlender is usability. The protocol should be able to operate within a normal torrent session, together with other normal peers who are unaware of the added anonymity layer [2].

The protocol achieves this by only making changes to the tracker servers and by adding the relay peers, Normal peers remain unchanged within the protocol. The normal peers also do not know if they are communicating with another normal peer or a relay peer [2]. So, this design goal is met.

### Resilio Sync and BitBlender.

BitBlender is a solution to provide anonymity for one-to-many file sharing within the BitTorrent protocol. Does it also provide anonymity when sharing through Resilio Sync? To answer this question we have to look at the differences between Resilio Sync and BitTorrent. First and foremost is the added authentication which means that a sender will always know who the receiver is as they have to manually confirm them to be legitimate session members. Adding relay peers in this setting seems useless as they would also need to be authenticated by the sending peer.

BitBlender does not seem suitable for Resilio Sync. The tracker and blender design would not work and if an adaption is made that they would work, one would simply end up with a similar protocol to the BitTorrent one.

**Table 2.** Design goals – Resilio sync vs BitTorrent vs BitBlender

Design goal #	Resilio Sync	BitTorrent	BitBlender
1. Receiver anonymity	-	-	+
2. Low computational resources	-	+	+
3. Easy to use	+	+	+
4. Free to use	-	+	+
5. Anonymity before TTP	-	+	+
6. Other privacy possible	+	+	+
7. Anonymity against adversary	-	-	+

As we can see in table 2 BitBlender meets all of our design goals and makes a good solution for our quest to receiver anonymity. However, we do have to investigate if BitBlender actually works as advertised.

### 5.1 Flaws with BitBlender.

There are some flaws with BitBlender, some of them are mentioned in the paper itself, but we also found some additional flaws.

#### Repeated pieces request.

Relay nodes and normal nodes are indistinguishable by behavior. So, an attacker does not know which node does what. However, the paper does note that there is an observable difference between the two nodes. A real node will not request the same piece twice, because as soon as it has obtained the requested piece, there is no need to request it again. A relay node can request the same piece twice or even more times, this scenario happens when two different active nodes request the same piece through the same relay node. This node can now be identified as a relay node.

The paper proposes the solution of normal nodes requesting pieces more than once. This is a well-studied solution within the traffic analysis field called cover traffic [2]. However, sending the same request multiple times does mean a reduction in performance of the protocol. If adding cover traffic is worth the performance reduction depends on what level of anonymity is desired.

The paper provides another solution in the form of caching requests. Such that it behaves more like a normal peer and will not have to request the same piece multiple times but can reply to a request from cache. This solution does not have a performance reduction but does require more resources from a relay peer. Relay peers also act more like normal peers as caching pieces can be seen equivalent to storing pieces and thus actually downloading the file.

**Legality.**

As Bauer et al. state in their paper, legally this is a difficult matter. Because the BitTorrent protocol is often used to spread copyrighted content<sup>17</sup> it is important to have a clear legal definition. Is a relay peer also illegal because one cannot distinguish it from a real node? Or is it aiding an illegal transaction? Or is it legal because the relay node itself only forwards requests? These are difficult matters without a definitive answer at this point and could be interesting for future work by someone with a background in law.

**Missing torrent identifier.**

The tracker sends a tuple  $(n, t)$  to the blender where  $n$  is the number of required relay nodes and  $t$  is unique identifier for the tracker. This creates a problem when a tracker hosts multiple torrent sessions (which is common in the BitTorrent protocol). A relay node that might join a session only gets  $p$  (the probability number upon to decide whether to join or not) and  $t$ . When relay nodes contact the server it does not know which torrent session to join.

**Traffic analysis attacks**

Using a timing correlation attack, adversaries might be able to learn information about which peers are relays and which are not. The timing of a sent request might reveal information about the path length and maybe if a peer is a relay or an active peer. If several peers are colluding, they might also be able to spot if a peer relays that request to another node or answers it directly.

**Lack of economic model.**

The paper by Bauer et al. relies on adding relay nodes to the normal set of nodes. The tracker requests them from the blender which makes the relay nodes join the torrent session. These relay nodes must spend computational resources in order to forward requests and appear as normal peers within the network. There is however no incentive for those relay nodes to do this.

If relay nodes are not compensated for joining a torrent session, then why would they?

In the next section solutions are proposed for the last three flaws.

---

<sup>17</sup> [https://en.wikipedia.org/wiki/Legal\\_issues\\_with\\_BitTorrent](https://en.wikipedia.org/wiki/Legal_issues_with_BitTorrent)

## 6 Solutions to the BitBlender flaws.

In this section several solutions to the aforementioned problems will be proposed. Some problems have trivial solutions such as the missing torrent identifier, but the incentive problem is less trivial and will be looked at from different viewpoints.

### **Solving missing torrent identifier.**

The solution to this problem is rather trivial. Instead of the tracker sending a tuple  $(n, t)$  to the blender it must send a 3-tuple that does not only contain the number of relay peers requested and the tracker identifier, but also a torrent identifier. For this torrent identifier the default identifier used by the BitTorrent protocol can be used; `info_hash`. The result would be the following 3-tuple:  $(n, t, i)$  where  $i$  is the `info_hash`.

The blender should then in its turn also forward  $t$  and  $i$  to the relay nodes together with the  $p$  as follows;  $(p, t, i)$ . Now relay nodes, when contacting the tracker to join a torrent session, also know which torrent session to join.

### **Mitigating traffic analysis attacks.**

Timing correlation attacks are a hard problem to mitigate without compromising on the performance of the system. This type of attacks is feasible on torrent sessions that contain a small number of peers, however, as the amount of peers increases the attacks is much harder to execute as the amount colluding peers necessary for this attack increases as well making it less feasible.

A possible mitigation strategy to the timing correlation attacks would be to introduce random delays at peers before answering or forwarding. This does, however, similar to adding cover traffic, negatively impact the performance. If this performance reduction is worth it, is again dependent on the desired level of anonymity.

### 6.1 Providing an incentive for relay peers.

Some sort of incentive should exist for relay peers to join a torrent session. This assures that relay peers are willing and motivated to join. If no incentive is provided to them, there is no real reason for them to join a torrent session as a relay, except to help others to stay anonymous. Relying on the goodwill of peers to join the session seems like a bad idea when attempting to make an anonymous system. Therefore, this section will attempt to give several different possible solutions on how to incentivize relay peers.

When brainstorming about this flaw it quickly became apparent that two main approaches can be taken to solve it:

- Reward system;
  - Rewarding a joined relay peer with something of value, for example each session a relay peer joins is rewarded with a monetary reward;
- Mandatory duty;
  - Making it mandatory to act as a relay node in order to act as a normal node in a torrent session. So, without relaying, normal participating is no longer allowed.

We are going to explore both possible solutions for incentivizing anonymous members. However, it is a difficult matter to solve because the peers must remain anonymous at all costs for the system to work. Whilst brainstorming the following issues arose:

- Both the tracker and blender do now know if a node is a relay node or not. To give them a reward or verify that it completed its mandatory duty, identifying which nodes are relays would be necessary, however, identification means some entity knows which nodes are which and therefore kills anonymity;
- In a system where nodes are not identified, a relay node may cheat by collecting a reward whilst not investing its computational resources;
- Traffic analysis on the rewards distribution would easily identify which peers are relays.

#### **Reward system.**

The reward system resembles a classical form of incentivizing as it is similar to how human employment works. They get payed for investing their time and effort. The reward system for relay peers attempts to achieve the same by rewarding relay peers for their service to the system.

The first problem we stumble upon is how to distribute rewards. This problem originates with the blender because it cannot tell which relay peers actually join the session and which do not in the BitBlender implementation. The blender only sends probability  $p$  to the peers and relies on the peers to decide whether to join or not.

There are two ways to tackle this problem:

- Identifying which nodes join the torrent session and rewarding that subset;
- Rewarding everyone in the blender regardless if they actually join or not with a reward (or partial reward).



Both of these methods are flawed unfortunately and would not work in a real world scenario.

In the first scenario the blender has to keep track of who it sent  $p$  to and afterwards has to verify them against the tracker list. This means the blender knows which are relay peers and which are not and therefore knows enough information about peers to kill anonymity within that session. Also, the blender model itself could then be simplified, because if it has to know which peers are relay anyway the probabilistic joining becomes redundant and the blender can just choose the peers itself. One of the strong points of BitBlender is that both the blender and tracker do not know which peers are relays which is functionality we do not want to alter as it is required for anonymity.

The second method is flawed because rewards are given to every peer. In some scenarios that might not be a problem but imagine a torrent tracker that requests a small set of relays from a blender that has a large set of relay nodes in it. Let  $n = 10$  for the requested relay nodes by the tracker. Let  $M$  be the set of relay nodes that the blender has where  $|M| = 10,000,000$ . In this scenario the rewards are disproportional to the computational resources that are obtained. Adjusting the rewards to be of lower value when this happens can create a scenario where it is not worth the effort for the relay peers to stay in the blender.

Another, more trivial, problem is the following; who pays for the rewards? BitTorrent is a free to use system that currently does not rely on exchanging things of value to download files. It is peer-to-peer and therefore there is no central authority to distribute the rewards. A possible solution might be to let all normal peers mine cryptocurrency whilst downloading and pay that to the blender, current yields are fairly low for that to be interesting and the adaptations to the protocol would be substantial. Also, a lot of computational resources are required which increase power usage significantly. BitTorrent's early and fast adoption when it was launched was attributed in part because it was free to use for everyone [18].

After this short dive into the reward system we found that it creates more problems than solutions, therefore we continue investigating by exploring the mandatory duty system that looks more promising. Thereafter we choose the most promising solution and propose an actual usable implementation for it.

### **Mandatory duty system.**

Another way to incentivize relays is to force normal peers to act as a relay before allowing them to join an actual desired session. This, again, is a hard problem to solve whilst keeping it a secret which peers are relays, and which are not within a torrent session.

The goal of this method is to only allow peers to participate in a torrent session if they already have contributed by acting as a relay in another session whilst the tracker, the blender and other peers cannot tell what type of peer it is joining.

Therefore we propose a new system of joining torrents by adding a token based authorization system. Joining a torrent session in BitTorrent is done by sending a request to the tracker. So, the tracker should be the authorizing party. To keep every peer anonymous the token will be an unidentifiable token. To join a torrent session the peer needs

to have a token that entitles them to join that session. The next problem we face is how to acquire a token if a peer is interested in downloading.

For a peer to be entitled to download a torrent it first has to contribute to the system as a relay node. The starting point for this new system is the blender. A peer joins its pool and waits for the first request for relays. It then receives probability  $p$  and calculates if it should join or not in normal fashion as described in BitBlender. This raises a problem for when the peer attempts to join as a relay because it does not yet have a token. To solve this, the blender will create a torrent specific token (a token that can only be used for a specific torrent session) for each possible relay and sends it together with  $p$  in the initial request.

This same system will be used for distributing tokens that entitle to a normal download. To allow this we introduce wildcard tokens. Wildcard tokens are similar to torrent specific tokens, but these allow to download any random torrent. These wildcard tokens are distributed only to peers that have acted as a relay.

## 6.2 Technical implementation for incentivizing relay nodes.

In Section 6.1 we explored possible solutions. The “mandatory duty solution” looks like the better candidate as it maintains one of the core values of BitTorrent, i.e. that it is free to use. However, when we dive deeper into both solutions we can see that they are in fact overlapping, both solutions are based on rewarding peers for their computational effort. The subtle difference is that the reward solutions requires something of value to transfer to the peer. The peer can join only to relay whilst not being interested in participating as a normal peer. Whilst in the mandatory duty solution only peers that desire to join the network will join, because their reward is worthless outside BitTorrent. In this section we will create a mandatory duty technical implementation for incentivizing relay peers.

After several revisions of the technical implementation for this incentive system for BitBlender in BitTorrent we found an implementation that works and described it in this section to get a clearer image of how this solution actually works. The technical implementation will be described for all participating parties; the blender, the tracker and the peers. Before that we start off with a list of design goals we want to achieve, a threat model and a short summary.

The requirements are:

- Fair token distribution – a peer should only get rewarded after relaying in a torrent session;
- Token integrity – a peer should not be able to cheat by creating tokens itself or modifying existing tokens;
- Preventing double spending – assure a peer cannot cheat by spending a token twice or more times;
- Maintain anonymity – the system should not compromise anonymity;
- Tracker may not find out what type of peer is joining the session;
- Blender should not be able to identify the peer that is joining a session;

### Threat model.

For our solution we will use the same threat model as BitBlender uses in their paper because it is used in numerous other low-latency anonymous networks and seems fitting to our implementation [2].

The threat model used by BitBlender is the following: “a non-global adversary that can participate in the BitBlender protocol as a colluding fraction of the total peers (either relay or normal)” [2] and adversaries can monitor the tracker list. We make another addition to this threat model; a tracker or blender can be malicious. However, we make the assumption that the blender and tracker do not collude as without this assumption it is hard to find a solid solution. Tracker and blender collusion will be investigated later in this section.

**Token incentivizing system in short.**

In this section we will describe the complete workflow of the new token system. For this workflow we assume a running environment with several torrent sessions. We will talk about initializing the system in a later section. If any check or verification fails, the protocol stops and the peer has to restart the process. For this explanation we use figure 1 on the next page. When, for example, '(3)' is used in the upcoming text we mean figure 1 sequence number 3.

Our starting point is a peer that wants to join a torrent session to download a file. To be able to do this, the peer first has to act as a relay peer. Therefore, the peer creates a serial number and applies a blinding factor to it (1). The peer can join a blender by sending a join request, containing the blinded serial number, to the blender (2). The peer now has to wait for the first blender request. Why a blinded serial number is needed and how blinding and unblinding works will be explained later.

The tracker sends a peer request, unchanged from BitBlender, to the blender with the amount of required peers, the tracker's own URL and the info\_hash of the torrent session (3).

Upon receiving a peer request from the tracker (3), the blender calculates probability  $p$  as normal in the BitBlender. However, it now also generates a new key pair for signing (4) for this specific torrent session. The blender then creates a token by signing the blinded serial number of the peer (4) and sends a join request to the peer containing the tracker URL, info\_hash and token (5). This token can only be used for this specific torrent session.

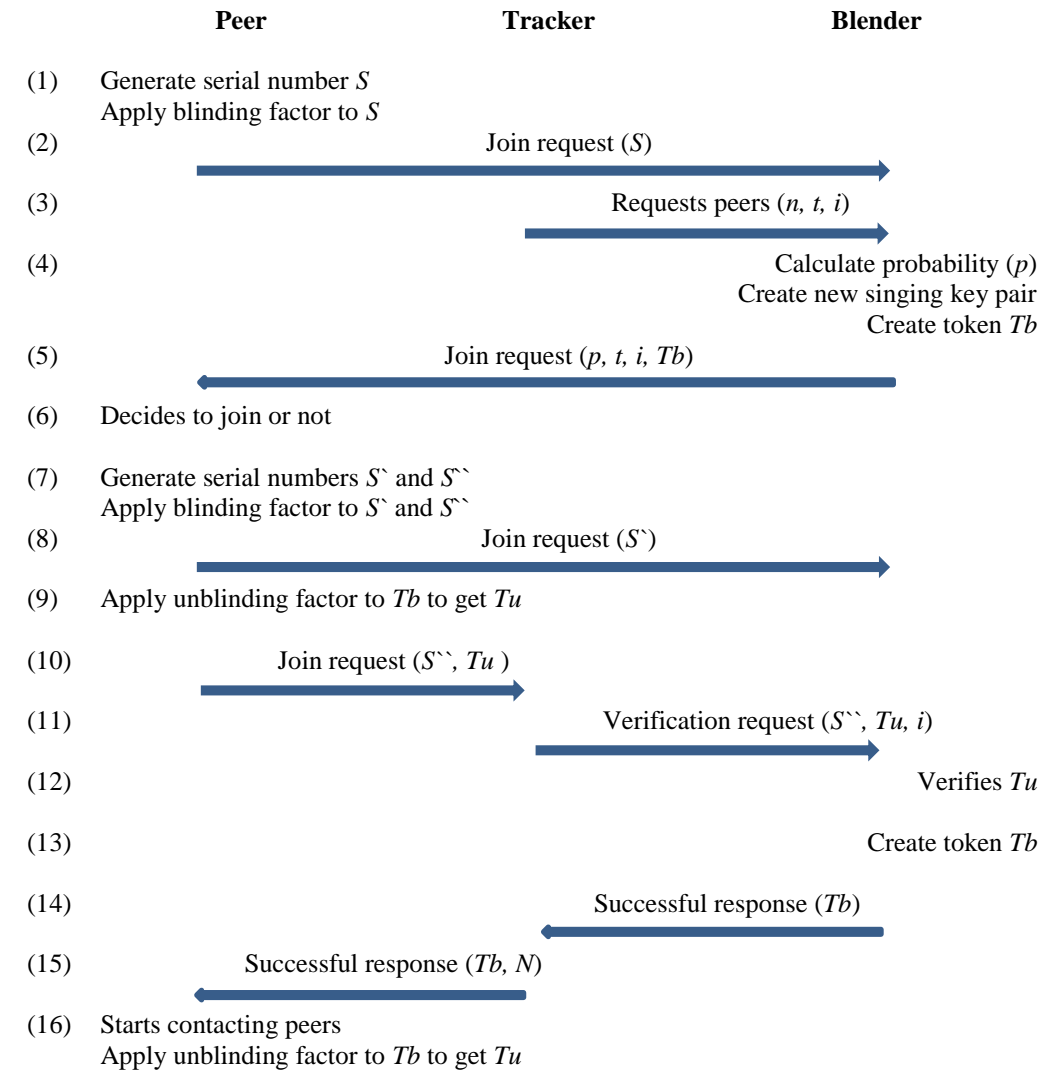
The peer now decides, according the normal BitBlender decision process, if it should join the torrent session or not (6). In any case, it creates a new blinded serial number (7) and sends that to the blender to indicate that it is ready for a new join request (8). If the peer decides to join, the peer unblinds the token (9) and sends that combined with a newly created blinded serial number (7) to the tracker to indicate that it wants to join the torrent session (10).

The tracker blindly forwards the token, new blinded serial number and the info\_hash of the torrent session to the blender (11). At this stage the tracker cannot learn if the peer is a relay or not. It only learns the signed token and a blinded serial number.

The blender verifies the token by checking the signature and checks if it is a new serial number that has not been used before (12). The token contains the unblinded serial number which is unequal to the blinded serial number the blender signed. Therefore the blender cannot link this token to a peer. The blender creates a new token by signing the new blinded serial number (13) with a different key pair and responds to the tracker with a successful state containing that new token (14).

The tracker receives the successful response with the new token from the blender and sends the new token with its normal BitTorrent response to the peer (15) who can then join the torrent session and unblind the token to get a new token that can be used for any torrent session (16).

Fig. 3. Token incentivizing protocol overview



$S$  = Serial number  
 $n$  = Amount of relay peers  
 $t$  = Tracker URL  
 $Tb$  = Token consisting of blinded  $S$   
 $Tu$  = Token consisting of unblinded  $S$   
 $N$  = normal response of BitTorrent containing peer list.

$i$  = info\_hash  
 $p$  = Probability

### **Preventing token misuse by using different keypairs.**

The blender prevents the peer from using the initial token for another, unauthorized, torrent session by using different key pairs for the tokens. Next to the keypair that is generated for the torrent session, the blender has two default key pairs; one for wildcard tokens and one for empty tokens. In (4) the blender uses the key pair that belongs to that info\_hash to sign the token. In (12) it uses that same key pair for verification. In (14) a new token is created, this is a wildcard token as a reward for relaying. This token is created using the default wildcard key pair.

If a peer wants to use its wildcard token, it creates a new blinded serial number. Then, steps 10 to 15 are executed. However, step 12 will fail as the blender tries to verify with the key pair belonging to the info\_hash. After this fail the blender will try the verification with the default wildcard keypair which will succeed. When this happens, the peer is not entitled to a new wildcard token, therefore the blender signs the new token with the empty keypair. If a peer attempts to use that empty token, both the info\_hash keypair and the wildcard keypair will fail the request will be denied.

This empty key is necessary to prevent the tracker from learning that the peer used a wildcard token which would happen if the blender was to return no token when a peer uses the wildcard token.

### **Peer.**

A peer has got some additional responsibilities with the new protocol. If it wants to join a blender, it will have to create a unique serial number  $S$ , blind it (1) and send that to the blender with the join request (2). Blinding is as simple as combining the serial number with a blinding factor.

When the blender sends request to join a torrent session, the peer will receive the following request:  $(t, i, p, T)$  (5).  $t$ ,  $i$  and  $p$  stand for tracker identifier, info\_hash and probability respectively.  $T$  is the token that entitles the peer to join that specific torrent session as a relay. This token  $T$  contains the blinded serial number. The peer must now unblind it before usage (9). Unblinding is done by removing the blinding factor. See the “preventing double spending with blind signatures” section for more information.

To join a torrent session as a relay peer, the peer now must include the token  $T$  in the join request to the tracker (10) together with a new blinded serial number.

The response of the tracker can be an error state (torrent session is not joinable) or a successful state (15). In the case of successful state, a new token is sent with it that can be used as a wildcard token for any random torrent session, allowing the peer to download a torrent without relaying.

When trading in this wildcard token the process is identical as described for a torrent specific token. Except for the fact that the tracker will now return an empty token which can be discarded. To join another session the peer has to restart the process and join a blender.

Peers cannot identify the type of the token, therefore the peer itself is responsible of keeping track of the type of the token. Empty tokens can be discarded upon reception as they hold no value.

### **Blender.**

The blender undergoes the biggest modifications as it will now be responsible for token creation and verification. We lay the creation and verification of the token with the blender to reduce overhead for the tracker and to keep all cryptographic actions in one place. The following addition to the blender have been made. The blender now has two default keypairs for signing; a wildcard keypair and an empty keypair. Next to those two default keypairs the blender creates a new keypair for each info\_hash it receives from the tracker when it receives a peer request (3).

After receiving this request, instead of only calculating the probability  $p$  and sending that to all relay peers that are in the blender set called  $B$ , the blender now must create a token for each peer in  $B$  (4). This token is created by signing the blinded serial number sent by the peer with a keypair that is generated for this torrent session specifically (4).

Let a torrent specific token be  $T$ . For each  $b_i$  in  $B$  a token  $T$  is created. To each  $b_i$  in  $B$  the blender sends the following data with the request ( $p, t, i, T$ ) (5). Where  $p$  is the probability,  $t$  is the tracker url,  $i$  is the info\_hash. So,  $T$  is added to the original Bit-Blender request.

#### *Token creation.*

A token consists of either a blinded serial number or an unblinded serial number. When creating the token, it consists of a blinded serial number.

To be able to differentiate between different types of tokens, the blender uses different keypairs for signing the different types. So using the correct keypair is necessary. For this example we want to create a torrent specific token and thus use the newly create signing keypair for that info\_hash with public key called  $pk$  and secret key called  $sk$ .

We call the blinded serial number that the blender received from the peer  $Sb$ .

A token  $T$  is created as follows:

$$T = \text{Sign}(bS, sk)$$

#### *Token verification.*

Between creation and verification of the token, the peer has unblinded the serial number. Thus the token now consists of an unblinded serial number we call  $Su$ .

Verification requests come from the tracker and have the following form: ( $i, T, Sb$ ).  $i$  is the info\_hash,  $T$  is the token and  $Sb$  is the new blinded serial number.

Verification of the signature happens with the blender's public key  $pk$  as follows:

$$S = \text{Verify}(Su, pk)$$

Due to the way blind signatures work, the signature is still valid even though the peer changed its content by unblinding the serial number. Blind signatures will be explained later in this section.

The blender attempts verification with the keypair belonging to the info\_hash that the tracker sent over with its request. This verification might fail. If this is the case the blender attempts verification with its default wildcard keypair. If that also fails the blender responds with an error state.

After the signature has been verified the blender has  $S$ . The blender starts by checking if  $S$  is already invalidated and it returns an error if so. If  $S$  is a unique serial number that the blender has not seen yet, it continues by sending a successful response.

The content of this successful response depend on which keypair was used. If the torrent specific info\_hash keypair was used the blender will create a new token signed with the wildcard keypair and send it with the response. If the wildcard keypair was used it will create a new token signed with the empty keypair and send that with the response.

After a successful state is sent, the blender invalidates  $S$  such that it cannot be used again after this transaction.

### Tracker.

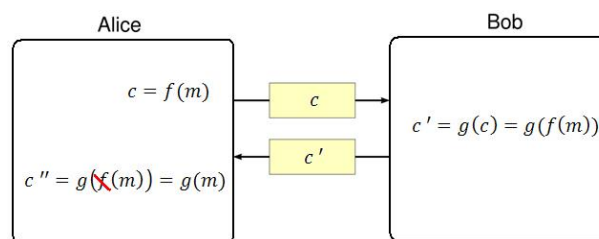
The tracker only has to facilitate the new authorization method, meaning it will have to receive the token and new serial number form the peer (10) and send it in a request with the info\_hash of the corresponding session to the blender for authorization (11). If the blender responds positively to the request (14) the peers request gets answered in normal fashion with the addition of a new token for the peer (15). If the blender responds negatively, the request is denied. The tracker does not need to know or care about the type of token, therefore protecting the anonymity of the peer.

### Preventing double spending with blind signatures.

To prevent double spending of tokens we use blind signatures as introduced by David Chaum [5] and used in e-cash protocols. Blind signatures work as follows:

The requesting party creates a message, blinds the message and send that blinded message to the signer. Blinding can be done by combining the message with a blinding factor. The signer signs the blinded message using a normal signing scheme and returns it to the requestor. The requestor can then unblind the message whilst keeping the signature valid when using for example RSA [5]. We do note, however, that RSA is not perfect and further research is required to choose a suitable signing method [3]. We will use blind signatures as an interactive protocol, not an actual implementation.

**Fig. 4.** Explanation of blind signatures<sup>18</sup>



<sup>18</sup> <https://upload.wikimedia.org/wikipedia/commons/f/f9/BlindSignature.jpg>



The implementation in our token system is as follows: the peer makes an unique identifier, a sort of serial number as used in cash money. In figure 4 this would be  $m$ . This serial number will be used to prevent double spending, meaning the blender will invalidate it upon spending. To validate this serial number the peer first blinds it serial number then sends it to the blender in figure 4 this blinding is done with function  $f(m)$ . The blender verifies if the peer is entitled to that signature, signs it and sends it back to the peer. In figure 4 the blender is Bob. The peer unblinds the serial number and uses it when joining a torrent session by sending it to the blender, in figure 4 this is done by removing  $f$ . The blender is responsible of validating that serial number, it does this by verifying that it has not seen this serial number before and invalidates it afterwards to assure it is not usable again.

Why go to all the trouble of using blind signatures? The answer is: to defend against a malicious blender. If we would let the blender create and distribute the serial numbers it can simply remember to what peer it send what serial number. If it sees that serial number again when validating it, it knows what peer it belongs to causing anonymity to fail.

#### **Alternative system for typing tokens**

We used different keypairs for differentiating between token types. However, one downside to this is that the blender has to keep track of all keypairs. If it serves a lot of torrent sessions, this might become computationally heavy. To solve this, we looked into a system that uses partially blind signatures. The serial number works the same and forms the part of partially blind signatures. The other part is an encrypted string containing the type of the token. This allows the token to be signed with only one key pair.

This solution can be found in appendix II but does need additional researching for the partial blind signatures themselves.

#### **Initializing the system.**

In our current configuration initializing the system results in a problem. Initialization means that there are no active torrent sessions and a new one is started. If a torrent wants to join that session it needs a token, to get a token it first needs to act as a relay peer. Meaning that it will join a blender as described in the system. However, a tracker will never request relay peers when its torrent session is empty. Resulting in a grid-locked system.

A possible solution to this problem is for the blender to give a peer  $x$  wildcard tokens upon joining the blender for the first time. In a real-world implantation of the system there are many blenders. So, a peer could cheat by joining a new blender each time and thus receiving wildcards without ever participating as relay node. A centralized list that keeps track of peers that have already received their “free” tokens might solve the problem but introduces a lot of unnecessary overhead.

A much simpler solution is available. Whenever a new torrent session is started, the tracker automatically requests  $x$  number of relays. Meaning that that session is properly started, new normal peers are surrounded by relay peers and thus anonymous from start. Those  $x$  relays now also have wildcard tokens and can join other sessions.

The latter of the two solutions is easy to implement and does a proper job of initializing the system.

#### **Efficiency problem when peers cheat.**

With this new system of incentivizing relay peers, a new problem arises. Peers gain benefit when joining a torrent session as relay in the form of a wildcard token. This means that peers are motivated to join as relays and might cheat. Cheating as a peer can be done as follows: when the blender requests peers to join a torrent session with a certain probability the peer can decide to always join regardless of the probability.

With the amount of relays, one might think the more the merrier. This is definitely true for anonymity. The anonymity set gets bigger resulting in better anonymity for participating normal peers. However, the flip side of this coin is deteriorating efficiency. Increasing relay peers results in increasing path lengths for messages between peers which makes the system decreasingly efficient.

Preventing this should be done by the blender. The blender knows the number of relay peers requested by the tracker and can impose an upper limit for relay peers as the blender knows which joining peers are normal or relaying. A hard-upper limit would result in leaking information about which peers are normal nodes. Because when the upper limit is reached all relays are refused and normal nodes are accepted. To mitigate this the blender could use a function to make accepting relay peers less likely the closer it gets to the upper limit.

The above solution to the efficiency problem introduces a new problem by itself, a fairness problem. Cheating nodes join immediately without calculating the join probability beating non-cheating nodes to the punch. Because of the scale of BitTorrent and the amount of peers using the protocol every day this is unlikely to become a problem in a real-world implementation. Also cheating peers are likely to have a goal, for example 4 torrents, after it has achieved that it will leave the system.

#### **Tracker and blender collusion.**

Anonymity cannot be achieved when a tracker and blender collude. If the tracker sends an identifying property of the peer to the blender in the token validation request, the blender can identify that peer to be either a relay or a normal peer and also send that knowledge back to the tracker.

A possible solution to this problem is to force all communication between tracker and blender to be public. This allows a peer to verify that no collusion takes place between the two before joining that blender or tracker. This solution is not ideal because it is hard to verify if the tracker and blender don't have a secret private communication channel. Therefore, this problem requires further research to find a better solution.

#### **Real-life scenario.**

Up until this point the incentivizing protocol has been described as if there is only one blender and one tracker. In a real world implantation, there are multiple trackers and multiple blenders. To make the protocol work in this scenario one simple modification

to the tokens is required. The URL of the blender should be added. That way the tracker can identify which blender to send the token validation request to.

**Malicious tracker or blender.**

Either the tracker or blender might be malicious. As an example, one can think of a blender that distributes invalid tokens or a tracker who refuses to accept a valid join request. A peer should be able to defend itself against this.

This is however a problem that is inherently solved by the BitTorrent protocol in the form of reputation. Because there are many trackers to choose from, if a tracker does not behave as expected it will get a bad reputation and eventually starve. The same goes for blenders added by BitBlender. If a tracker behaves maliciously, it will starve. If the tracker behaves as normal but the blender is malicious, the tracker will still starve. So, the tracker needs to select a new blender to be able to survive. For a peer it is irrelevant which party is malicious if a request is not handled properly it can choose to leave and select a new tracker or blender.

## 7 Conclusion

This thesis originated from the wish to have receiver anonymity within file sharing. To realize this wish we started this thesis by looking at definitions for file sharing, architectural models for possible solutions, existing solutions and what privacy the protocol should offer. For our goal, receiver anonymity, the peer-to-peer architecture came out as the clear winner for our protocol, as we expected. Self-hosted solutions relies too much on users having knowledge and resources and for the server client model we have to trust large companies. We found that there are no ready-to-use solution in existence that offer any form of anonymity for file sharing.

After looking at the most promising existing solution we found Resilio Sync, which offers a solid solution for private file sharing and synchronizing by adding encryption and authentication to its underlying protocol, BitTorrent, but offers no anonymity.

BitTorrent, however, provided a solid starting point for an existing paper called BitBlender. BitBlender achieves anonymity in BitTorrent by using the crowds protocol. Although anonymity is achieved with BitBlender it does have several flaws which needed to be resolved/mitigated. That is where the contribution of this thesis comes in. We proposed solutions to the several problems, the biggest was the lack of incentives for relay peers.

To solve the lack of incentives we proposed a token based protocol reliant on attribute based credentials and blind signatures. With this new token system, peers have to act as a relay in a random torrent session before they can act as a normal peer in their desired torrent session, Peers are rewarded a token when relaying that can be used to download a random torrent of their choosing.

So, we ended up with a protocol that protocol meets all the design goals that are defined in section 3, is based on BitTorrent, uses BitBlender to add anonymity and implements a newly proposed token system to incentive relay peers within BitBlender.

## 8 References

1. Abe, M., & Okamoto, T. (2000). Provably secure partially blind signatures. In Annual International Cryptology Conference (pp. 271-286). Springer, Berlin, Heidelberg.
2. Bauer, K., McCoy, D., Grunwald, D., & Sicker, D. (2008). BitBlender: Light-weight anonymity for BitTorrent. In Proceedings of the workshop on Applications of private and anonymous communications (p. 1). ACM.
3. Boneh, D. (1999). Twenty years of attacks on the RSA cryptosystem. Notices of the AMS, 46(2), 203-213.
4. Brinkmann, M. (2016). BitTorrent Sync now called Resilio Sync - gHacks Tech News. Retrieved May 11, 2019, from <https://www.ghacks.net/2016/06/16/bittorrent-sync-resilio-sync/>
5. Chaum, D. (1983). Blind signatures for untraceable payments. In Advances in cryptology (pp. 199-203). Springer, Boston, MA.
6. Cohen, B. (2003). Incentives build robustness in BitTorrent. In Workshop on Economics of Peer-to-Peer systems (Vol. 6, pp. 68-72).
7. Cohen, B. (2008). The BitTorrent Protocol Specification. Retrieved May 30, 2019, from [http://www.bittorrent.org/beps/bep\\_0003.html](http://www.bittorrent.org/beps/bep_0003.html)
8. Cox, J. (2016). Hackers Stole Account Details for Over 60 Million Dropbox Users - Motherboard. Retrieved March 28, 2019, from [https://motherboard.vice.com/en\\_us/article/nz74qb/hackers-stole-over-60-million-dropbox-accounts](https://motherboard.vice.com/en_us/article/nz74qb/hackers-stole-over-60-million-dropbox-accounts)
9. Davies, R. (2018). Apple becomes world's first trillion-dollar company . Retrieved June 5, 2019, from <https://www.theguardian.com/technology/2018/aug/02/apple-becomes-worlds-first-trillion-dollar-company>
10. Dingledine, R., Mathewson, N., & Syverson, P. (2004). Tor: The second-generation onion router. Naval Research Lab Washington DC.
11. Drago, I., Mellia, M., M Munafo, M., Sperotto, A., Sadre, R., & Pras, A. (2012). Inside dropbox: understanding personal cloud storage services. In Proceedings of the 2012 Internet Measurement Conference (pp. 481-494). ACM.
12. Farina, J., Scanlon, M., & Kechadi, M. T. (2014). Bittorrent sync: First impressions and digital forensic implications. Digital Investigation, 11, S77-S86. <https://www.sciencedirect.com/science/article/pii/S1742287614000152>
13. Jones, B. (2008). How to Make the Best Torrents - TorrentFreak. Retrieved July 19, 2019, from <https://torrentfreak.com/how-to-make-the-best-torrents-081121/>
14. Jones, B. (2008). Trading BitTorrent Tracker Invites , Commodity or Curse? . Retrieved May 30, 2019, from <https://torrentfreak.com/trading-bittorrent-tracker-invites-080115/>
15. Keach, S. (2019). Apple iCloud bug 'let ANYONE read your private iPhone notes' – and was 'kept a secret', security expert claims. Retrieved June 5, 2019, from <https://www.thesun.co.uk/tech/8313049/iphone-icloud-breach-bug/>
16. Kincaid, J. (2011). Dropbox Security Bug Made Passwords Optional For Four Hours. Retrieved March 28, 2019, from <https://techcrunch.com/2011/06/20/dropbox-security-bug-made-passwords-optional-for-four-hours/>
17. Konstantin (2018). What's the difference between peer to peer and client server? - Resilio Blog. Retrieved March 28, 2019, from <https://www.resilio.com/blog/whats-the-difference-between-peer-to-peer-and-client-server>
18. Legout, A., Urvoy-Keller, G., & Michiardi, P. (2005). Understanding bittorrent: An experimental perspective.

19. Lindberg, O. (2018). UX Evolutions: How Dropbox Has Changed Over the Last 10 Years. Retrieved March 20, 2019, from <https://theblog.adobe.com/ux-evolutions-dropbox-changed-last-10-years/>
20. Miltenburg, O. (2015). Synology dicht ernstige lekken in Download- en Video Station-software. Retrieved April 16, 2019, from <https://tweakers.net/nieuws/105191/synology-dicht-ernstige-lekken-in-download-en-video-station-software.html>
21. Okamoto, T. (2006). Efficient blind and partially blind signatures without random oracles. In *Theory of Cryptography Conference* (pp. 80-99). Springer, Berlin, Heidelberg.
22. Orosz, A. (2018). Cloudlessly Connected: Stay in Sync With Resilio. Retrieved April 1, 2019, from <https://www.wayoflinux.com/blog/cloudless-file-sync>
23. Pfitzmann, A., & Hansen, M. (2010). A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management.
24. Pfitzmann, A., & Waidner, M. (1987). Networks without user observability. *Computers & Security*, 6(2), 158-166.
25. Reiter, M. K., & Rubin, A. D. (1998). Crowds: Anonymity for web transactions. *ACM transactions on information and system security (TISSEC)*, 1(1), 66-92.
26. Scanlon, M., Farina, J., & Kechadi, M. T. (2015). Network investigation methodology for BitTorrent Sync: A Peer-to-Peer based file synchronisation service. *Computers & Security*, 54, 27-43.
27. Snelling, D. (2018). The Pirate Bay BLOCKED - Millions stopped from accessing torrent site in tough new ban. Retrieved June 18, 2019, from <https://www.express.co.uk/life-style/science-technology/905931/The-Pirate-Bay-block-ban-torrent-downloads-ISP>
28. Toole, R., & Vokkarane, V. (2006). Bittorrent architecture and protocol. University of Massachusetts Dartmouth, Dartmouth.
29. US SEC. (2018). Dropbox, Inc.. Retrieved March 28, 2019, from <https://www.sec.gov/Archives/edgar/data/1467623/000119312518055809/d451946ds1.htm>
30. vd Spek, O. (2008). UDP Tracker Protocol for BitTorrent. Retrieved June 27, 2019, from [http://www.bittorrent.org/beps/bep\\_0015.html](http://www.bittorrent.org/beps/bep_0015.html)
31. Wallen, J. (2016,). Speculations on why ownCloud's founder forked its popular product into Nextcloud. Retrieved April 1, 2019, from <https://www.techrepublic.com/article/owncloud-founder-has-forked-their-product-into-nextcloud/>
32. Warren, S. D., & Brandeis, L. D. (1890). Right to privacy. *Harv. L. Rev.*, 4, 193
33. White, C. (2011). Dropbox can legally sell all of your files [Update]. Retrieved March 28, 2019, from <https://www.neowin.net/news/dropbox-legally-owns-all-of-your-files>
34. Winder, D. (2019). Data Breaches Expose 4.1 Billion Records In First Six Months Of 2019. Retrieved August 21, 2019, from <https://www.forbes.com/sites/davey-winder/2019/08/20/data-breaches-expose-41-billion-records-in-first-six-months-of-2019/>

## 9 Appendix I: Existing solutions

### 9.1 Testing framework

To get consistent and comparable results we define a testing framework which will be used when looking into existing solutions.

#### Framework

What	Contains	Why
Short introduction	Who provides the service, what model does it use?	To introduce the reader to the solution under investigation.
Business model	Is the service free to use? Does it have tiers in a subscription model?	To learn how these companies create a sustainable solution and to find out if a free model is possible.
Functionality	What functionality is offered by the service?  Backup: Synchronization: Sharing: <hr/> Versioning: Web interface: Applications: Offline use:	To learn what functionality can be combined within a service and how they integrate with each other.
User experience	This will be a subjective part with the impression of the author that lists benefits and drawbacks of the UX.	One of the hardest parts of creating such a solution is to keep it simple and easy for the user. We hope to learn how existing solution achieve this.
How does it work	Install method: How to back up a file: How to sync a file: How to retrieve a file: How to share a file: Possibility to sync extra folders:	To learn how these solutions overcome specific problems.
Privacy	Offers anonymity: Does this service respect its user's privacy? Does it offer anonymity?	In the introduction we assumed that privacy is an issue with existing solutions, is this really the case?
Security	Is the security of the service in order? Have there been any data breaches?	What security measures are being taken, are they sufficient or is more required.
Remainders	Here we mention interesting things that do not fit in other segments.	To be complete and not miss any relevant information.

Conclusion	What are the most interesting or valuable concepts that are usable for this thesis.	-
------------	---	---

## 9.2 Existing solutions

### Server-client model services.

#### *Dropbox.*

<b>Introduction</b>	
Dropbox <sup>19</sup> is offered by Dropbox Inc as a service, it has more than 500 million users of which more than 11 million have a paid subscription [29]. Dropbox uses the client-server model but also uses p2p techniques to speed up the syncing of files that are already available on other devices that are accessible through LAN.	
<b>Business model</b>	
Dropbox has a freemium <sup>20</sup> model in which users get 2gb of free storage but are able to get more free storage through referrals. To get more storage and/or functionality users have to pay a subscription fee <sup>21</sup> .	
<b>Functionality</b>	
Backup:	no
Synchronization:	yes
Sharing:	yes
Versioning:	yes (30 or 120 days depending on subscription)
Web interface:	yes
Applications:	yes
Offline use:	yes

<sup>19</sup> <https://www.dropbox.com>

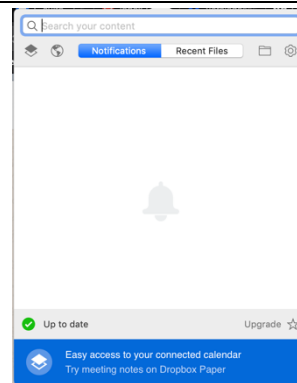
<sup>20</sup> Freemium is a business model where the service is offered for free, but offers extra features for a price. Freemium is a combination of the words free and premium.

<sup>21</sup> Choose the right Dropbox for you and your business - <https://www.dropbox.com/plans?trigger=nr>



### User experience

The user experience that Dropbox offers is very good. Dropbox offers different applications for a wide variety of platforms, mobile and desktop, with a strong and very simple to use UI. In essence, after downloading and installing the application, only a login is required and the user is ready to use Dropbox [19]. The GUI and UX are easy to understand and practical in its use. Both the website and application offer nearly all functionality. The website offers document collaboration which the application does not.



#### Benefits:

- Both client and website offer a complete functionality package. No need to go to the website to share a file for example;

#### Drawbacks:

- Limitations in the preferences of the client, only one folder can be synced.
- Aggressive integrations with other programs such as Microsoft Word.
- Not possible to use files without syncing them first.

### How does it work?

#### Install method:

Dropbox can be installed to a pc locally via an installer, but it can also be used through their website without the need to download.

#### How to sync a file:

Dropbox syncs one specific folder on the users pc. So to sync a folder an user has to place the file within that folder. Files can also be uploaded to their website manually.

#### How to retrieve a file:

Files can be downloaded from the website, but will also be synced into the selected folder.

#### How to share a file:

A file can be shared both from the installed client and the website. To share a file, the email address of the recipient can be used. Or a shareable link<sup>22</sup>. The recipient is able to view the file on the website and download it from there without having an account.

#### Possibility to sync extra folders:

This is not possible with Dropbox. It is however possible to select which folders to sync within the main Dropbox folder.

<sup>22</sup> Link means URL

<p><b>Privacy</b></p> <p>Offers anonymity: no</p> <p>The privacy of Dropbox does leave a lot to desire. When we look into privacy we also look at privacy breaches. A couple of examples:</p> <ul style="list-style-type: none"> <li>• Whilst accounts were protected by passwords, authentication without password was still possible due to a bug [16]</li> <li>• Dropbox claims ownership of all files uploaded to their service [33]</li> <li>• Over 60 million user account details compromised [8]</li> </ul>
<p><b>Security</b></p> <p>Dropbox provides a clear whitepaper that explains all security related matters<sup>23</sup>.</p> <p>Encryption</p> <p>Transit: 128-bit AES, TLS</p> <p>End-to-end: no</p> <p>Cloud storage: yes 256 bit AES</p> <p>Authentication:</p> <p>2 factor: yes</p> <p>Type: Name and password</p> <p>Tracking: Claims to not sell to third parties</p>
<p><b>Reminders</b></p> <p>LAN sync, a way to speed up synchronization by first checking via LAN for the file, but a connection with the Dropbox servers is required [11]. This helps when internet speeds are slow and also reduces load on the Dropbox servers.</p>
<p><b>Conclusion</b></p> <p>Dropbox offers a very solid service for file backup and sharing, but their service does come with privacy concerns. The most interesting technique utilized by Dropbox is LAN Sync which might be useful to investigate further. The UX offers a easy to use system that also offers insight into what users want and how they use the system.</p>

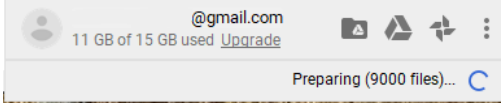
### *Drive.*

<p><b>Introduction</b></p> <p>Drive<sup>24</sup> is a service offered by Google for consumers and business users. For business users it is bundled with G Suite or available as Drive Enterprise<sup>25</sup>. Drive uses the client-server model.</p>
--

<sup>23</sup> Dropbox Business Security. Retrieved April 15, 2019, from [https://aem.dropbox.com/cms/content/dam/dropbox/www/en-us/business/solutions/solutions/dfb\\_security\\_whitepaper.pdf](https://aem.dropbox.com/cms/content/dam/dropbox/www/en-us/business/solutions/solutions/dfb_security_whitepaper.pdf)

<sup>24</sup> <https://www.google.com/drive/>

<sup>25</sup> Take Drive to work - [https://gsuite.google.com/intl/en\\_us/driveforwork](https://gsuite.google.com/intl/en_us/driveforwork)

<b>Business model</b>	
Drive uses a freemium model for consumers similar to the one Dropbox uses. 15gb is free, if users need more storage they can upgrade to Google One <sup>26</sup> . The business version of Drive does not have a free tier <sup>25</sup> .	
<b>Functionality</b>	
Backup:	yes
Synchronization:	yes
Sharing:	yes
Versioning:	yes (30 days or 100 versions, option to keep forever)
Web interface:	yes
Applications:	yes
Offline use:	yes
<b>User experience</b>	
<p>User experience of Drive is, as to be expected from Google, very good. User interface of the different applications is subjectively very good. Operating system support is covered by a variety of different applications.</p> <p>Benefits:</p> <ul style="list-style-type: none"> <li>- Easy to use,</li> <li>- Client offers a lot of functionality which decreases the necessity to use the site.</li> </ul> <p>Drawbacks:</p> <ul style="list-style-type: none"> <li>- Not possible to use files without syncing them first.</li> </ul>	
<b>How does it work?</b>	
<p><b>Install method:</b> Drive can be installed to a pc locally via an installer, but it can also be used through their website without the need to download.</p> <p><b>How to backup a file:</b> During installation, or in the settings menu, the user can select which folders they want to backup continuously. After configuring the backup, it is fully automated.</p> <p><b>How to sync a file:</b> Drive syncs one specific folder on the users pc. So to sync a folder an user has to place the file within that folder. Files can also be uploaded to their website manually.</p> <p><b>How to retrieve a file:</b></p>	

<sup>26</sup> Google One - More storage and extra benefits from Google - <https://one.google.com/about>

<p>Files can be downloaded from the website, but will also be synced into the selected folder.</p> <p><b>How to share a file:</b> A file can be shared both from the installed client and the website. To share a file, the email address of the recipient can be used. Or a shareable link. The recipient is able to view the file on the website and download it from there without having an account.</p> <p><b>Possibility to sync extra folders:</b> This is not possible with Drive. It is however possible to select which folders to sync within the main Drive folder.</p>										
<p><b>Privacy</b></p> <p>Offers anonymity: no The privacy of Drive is interwoven with that of Google because they do not have a specific privacy policy for Drive but instead a reference to the Google general Privacy Statement<sup>27</sup>, therefore we look into Google's privacy in general. Google has the capability to mine all the data and metadata it receives and use that to create advanced user profiles to help them to tailor services and advertisements to specific users. We will not go into great detail here because reliable sources are hard to find but we can say at least that Google has access to all files that are stored on Drive.</p>										
<p><b>Security</b></p> <p>Google provides a clear whitepaper that explains all security related matters<sup>28</sup>.</p> <p>Encryption</p> <table> <tr> <td>Transit:</td> <td>yes (see whitepaper for more information)</td> </tr> <tr> <td>End-to-end:</td> <td>no</td> </tr> <tr> <td>Cloud storage:</td> <td>yes 128 bit (or stronger) AES (except videos)</td> </tr> </table> <p>Authentication:</p> <table> <tr> <td>2 factor:</td> <td>yes</td> </tr> <tr> <td>Type:</td> <td>Name and password</td> </tr> </table> <p>Tracking: Claims to not sell to third parties</p>	Transit:	yes (see whitepaper for more information)	End-to-end:	no	Cloud storage:	yes 128 bit (or stronger) AES (except videos)	2 factor:	yes	Type:	Name and password
Transit:	yes (see whitepaper for more information)									
End-to-end:	no									
Cloud storage:	yes 128 bit (or stronger) AES (except videos)									
2 factor:	yes									
Type:	Name and password									
<p><b>Reminders</b></p> <p>-</p>										
<p><b>Conclusion</b></p> <p>Drive offers a very solid service for file backup, synchronization and sharing. Although it is similar to the service Dropbox provides, it does have the welcome addition of backups. Lessons can be learned in the ease of usability of the service and</p>										

<sup>27</sup> Google Drive Terms of Service - Google Drive Help. <https://support.google.com/drive/answer/2450387?hl=nl>

<sup>28</sup> How Google Uses Encryption to Protect Your Data - [http://services.google.com/fh/files/helpcenter/google\\_encryptionwp2016.pdf](http://services.google.com/fh/files/helpcenter/google_encryptionwp2016.pdf)

Drive proves that one service can offer all three actions without compromising usability.

### *WeTransfer.*

#### **Introduction**

WeTransfer<sup>29</sup> does not offer the synchronization or backup of data like the other services we looked at so far do, but it is purely a file sharing site. It has only one goal and that is to transfer files between users. Account creation is not necessary for either sending or receiving party. WeTransfer uses the client-server model.

#### **Business model**

WeTransfer offers a free plan which allows users to send up to 2gb. If users need more storage than 2gb then they will have to pay for WeTransfer Plus which allows up to 20gb per transfer and offers some additional personalization functionality<sup>30</sup>.

#### **Functionality**

Backup:	no
Synchronization:	no
Sharing:	yes
Versioning:	no
Web interface:	yes
Applications:	yes
Offline use:	no

#### **User experience**

The user experience is very good. It is simple to use with a very solid user interface. Because WeTransfer only offers one service, namely file sharing, there is no clutter or confusion on the service.

##### Benefits:

- Extremely easy: drag and drop files, fill in email and click send
- No account needed for sending or receiving

##### Drawbacks:

- Very limited functionality, only file sharing is possible.

#### **How does it work?**

##### Install method:

WeTransfer operates primarily via a website but can be installed to a device locally via an installer.

##### How to retrieve a file:

<sup>29</sup> <https://wetransfer.com/>

<sup>30</sup> WeTransfer Plus - <https://wetransfer.com/plus>

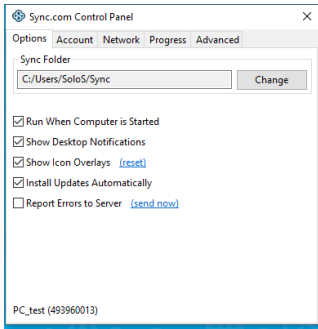
Files can be downloaded from the website through a link
<p>How to share a file:  A file can be shared both from the installed client and the website. To share a file, the email address of the recipient can be used or a link. The recipient is able to download it from the website without having an account.</p>
<b>Privacy</b>
<p>Offers anonymity: no  When looking into privacy it is interesting to know if WeTransfer has access to their users' files. They have, because in a statement about their takedown policy they say the following: "WeTransfer finds or is notified of unlawful files or content being distributed through it is Service"<sup>31</sup>, which suggests that they are very capable of looking into users files and even do so to comply with the law. WeTransfer unfortunately does not provide any sort of technical documentation.</p>
<b>Security</b>
<p>Unfraternally, unlike Dropbox and Drive, WeTransfer does not offer a whitepaper. The following information is from a FAQ section<sup>32</sup></p> <p>Encryption</p> <p>Transit:           yes, TLS  End-to-end:       no  Cloud storage:   yes 256 bit AES</p> <p>Authentication:</p> <p>2 factor:           not applicable  Type:               none</p> <p>Tracking:           Claims to not sell to third parties</p>
<b>Reminders</b>
-
<b>Conclusion</b>
<p>In conclusion, WeTransfer offers a very good user experience for file sharing. It does one action, file sharing, extremely well. But it misses a distinguishing factor from Dropbox and Drive. They offer the same ease of file sharing, but without the downside the other two have that the sender needs an account.</p>

<sup>31</sup> Notice and Take Down Policy - <https://wetransfer.com/legal/takedown>

<sup>32</sup> How secure is your platform? - <https://wetransfer.zendesk.com/hc/en-us/articles/210092453-How-secure-is-your-platform>

Up until this point, we saw three very good solutions above, of which Drive is the most complete regarding functionality. However these solutions are all from large corporations who do not have a good track record regarding privacy/security breaches in the news. We will now divert our attention to encrypted and privacy friendly storage. In our research we came up with the following, non-exhaustive, list: Sync, Spideroak, Mega, Tresorit, pCloud, AllSync . Looking at all of these services would be very time intensive and redundant as they offer similar functionality, therefore we selected two namely: Sync and Mega.

*Sync.*

<b>Introduction</b>	
Sync <sup>33</sup> claims to be the secure Dropbox alternative by offering end-to-end encryption, no tracking and global data privacy compliance <sup>34</sup> . It offers synchronization and sharing but no backup, other than the vault (explained below).	
<b>Business model</b>	
Sync has a freemium model in which users get 5gb of free storage but are able to get more free storage through tasks. To get more storage and/or functionality users have to pay a subscription fee <sup>28</sup> .	
<b>Functionality</b>	
Backup:	no
Synchronization:	yes
Sharing:	yes
Versioning:	yes (free has 30 days, paid has unlimited)
Web interface:	yes
Applications:	yes
Offline use:	yes
<b>User experience</b>	
<p>The user experience that Sync offers is very good. Sync offers different applications for a wide variety of platforms, mobile and desktop, with a strong and very simple to use UI. In essence, after downloading and installing the application, only a login is required and the user is ready to use Sync. The GUI and UX are easy to understand and practical in its use. Both the website and application offer nearly all functionality. The website offers document collaboration which the application does not.</p>	
	

<sup>33</sup> <https://www.sync.com/>

<sup>34</sup> a Secure Dropbox Alternative. - <https://www.sync.com/secure-dropbox-alternative/>

<p><b>How does it work?</b></p> <p><b>Install method:</b> Sync can be installed to a pc locally via an installer, but it can also be used through their website without the need to download.</p> <p><b>How to sync a file:</b> Sync syncs one specific folder on the user's pc. So to sync a folder an user has to place the file within that folder. Files can also be uploaded to their website manually.</p> <p><b>How to retrieve a file:</b> Files can be downloaded from the website, but will also be synced into the selected folder.</p> <p><b>How to share a file:</b> A file can be shared both from the installed client and the website. To share a file, the email address of the recipient can be used. Or a shareable link. The recipient is able to view the file on the website and download it from there without having an account.</p> <p><b>Possibility to sync extra folders:</b> This is not possible with Sync. It is also not possible to select which folders are synced. Everything is synced except for the vault (explanation below)</p>
<p><b>Privacy</b></p> <p>Offers anonymity: no Sync.com presents itself as a “privacy over profit” company<sup>35</sup>. It is compliant with all major privacy laws. When looking into their security privacy seems well taken care of by their encryption mechanisms. No major data breaches have been found whilst searching the web.</p>
<p><b>Security</b></p> <p>Encryption  Transit: 2048-bit RSA, SSL/TLS encryption  End-to-end: yes  Cloud storage: yes</p> <p>Authentication:  2 factor: yes  Type: Name and password</p> <p>Tracking: No third-party tracking</p>
<p><b>Reminders</b></p> <p>Sync Vault: Sync offers the vault as a cloud only storage based solution. So if a user wants to back up a file without it syncing to all its devices it can put it into the vault. It can be downloaded from the web interface when necessary</p>

<sup>35</sup> <https://www.sync.com/blog/your-privacy-matters/>



**Conclusion**

Sync offers a very similar experience to Dropbox, but it makes a stronger security and privacy argument in its marketing. This argument proves that secure and privacy friendly storage is possible. End-to-end encryption, encryption in transit and encryption in storage is interesting!

*Mega.*

After researching Mega, it became apparent that it is very similar to Sync, but additionally it offers backup. We decided not to research it in depth because of its similarities to Sync.

**9.3 Peer-to-peer model**

The major problem with the client-server model is the necessity to trust a third party for data storage. Their security has to be in good order but they also have to respect their users privacy. Both are difficult to guarantee without blindly trusting what the companies behind the services say. We now take a look into services that offer p2p based syncing and/or sharing.

When looking for services that utilize p2p we found solutions that requires user tinkering and thus are not easily ready to use. We will not look at the following solutions in detail for that reason: Librevault (in alpha), Unison, Syncany (in alpha), dat:// (requires command line), Tahoe-LAFS, HRcloud2, Xamiro and Filstash.

*Resilio Sync.***Introduction**

Resilio sync<sup>36</sup> is the first service in this list that does not offer a client-server model but instead offers a solution based on the p2p model [17]. So, in essence, Resilio offers the software to enable its users to use p2p technology.

**Business model**

Resilio Sync offers several versions of their software with differing pricing. There is a free version, a pro version for consumers and several options for teams and businesses. Each version offers its own set of functionality where paid versions offer more functionality than free services<sup>37</sup>.

**Functionality**

Backup:	yes
Synchronization:	yes
Sharing:	yes

<sup>36</sup> <https://www.resilio.com/>

<sup>37</sup> Resilio Sync Plans. - <https://www.resilio.com/individuals/>

Versioning:	yes
Web interface:	no
Applications:	yes
Offline use:	yes
<b>User experience</b>	
<p>The user experience that Resilio offers is good in terms of application availability on several operating systems and user interface. However, due to the nature of p2p solutions, the peers have to be online since there is no central server. So, if a peer is down, it will not be synced and it will not be synced from.</p>	
<b>How does it work?</b>	
<p><b>Install method:</b>          Downloading and installing the Resilio Sync client. No web gui is available due to the P2P technology making that very difficult to build.</p> <p><b>How to sync a file:</b>          By creating a standard folder in the client, creating a read only or write key and sending it to the other client. The other client pastes the link in their client and the folder is synced</p> <p><b>How to retrieve a file:</b>          Not possible. All files are synced at all times, deleting a file means it will be deleted everywhere.</p> <p><b>How to share a file:</b>          Similar to synchronization, but now the user should select the read only key and a time limit for the file to be shared.</p> <p><b>Possibility to sync extra folders:</b>          Yes, completely up to the user, no limits imposed here.</p>	
<b>Privacy</b>	
<p><b>Offers anonymity:</b> no          When diving into privacy with Resilio, it has a trivial major advantage over the other services we look at so far. It does not store user data on a central server. So Resilio does not have access to user data at all. The only point of interference Resilio has is their relay server<sup>38</sup>. The relay server has the capability to log metadata about users syncing behaviour. Resilio does report that data that travels through the relay server is encrypted but does not make any claims about metadata. The relay server is, however, optional [22].</p>	

<sup>38</sup> What ports and protocols are used by Sync? - <https://help.resilio.com/hc/en-us/articles/204754759-What-ports-and-protocols-are-used-by-Sync->

<b>Security</b>	
Encryption	
Transit:	yes
End-to-end:	yes
Storage:	yes
Authentication:	
2 factor:	no
Type:	keys
Tracking:	No tracking whatsoever
<b>Reminders</b>	
Self-owned nodes	
<b>Conclusion</b>	
In conclusion, Resilio offers a solid p2p software service that can be purchased with a one-time payment instead of a subscription model. For this paper, Resilio forms an interesting look into the possibilities and technologies behind p2p file syncing and sharing. Missing anonymous file sharing is a shame, but might be resolvable by adapting the underlying protocol.	

#### 9.4 Self-hosted model

A final model we can take a look at is the self-hosted model. Within this model one of the other two models is used. For example a user creates its own server within its LAN thus creating a client-server model. Or, users utilize a technique such as AFS to create their own p2p system.

*OwnCloud / NextCloud.*

<b>Introduction</b>
Owncloud <sup>39</sup> and NextCloud <sup>40</sup> are similar offerings that make it possible to implement a client-server model where the server is owned by the user itself. This does require more know-how of the technical side as it is required to have a webserver to run OwnCloud or NextCloud on it. This is however an interesting value proposition, and therefore will be researched <sup>41</sup> . We will only look into OwnCloud but note that NextCloud is very similar as it is a fork of OwnCloud [31].

<sup>39</sup> <https://owncloud.org/>

<sup>40</sup> <https://nextcloud.com/>

<sup>41</sup> ownCloud Feature Overview & Possibilities. - <https://owncloud.org/features/>

<b>Business model</b>	
Both Owncloud and NextCloud are open source and free to use but do offer subscriptions for example for enterprise solutions <sup>42</sup> These enterprise subscriptions also offer additional functionality such as “File Firewall”.	
<b>Functionality</b>	
Backup:	no
Synchronization:	yes
Sharing:	yes
Versioning:	yes
Web interface:	yes
Applications:	yes
Offline use:	yes
<b>User experience</b>	
Offers anonymity: no The user experience offered by OwnCloud feels a lot like the other services we looked at. It looks good, works logical and fast. It does not feel like open-source often does. It offers many clients with a well thought out design. When installing OwnCloud onto a server it has an easy to use setup wizard to guide users through the process. However due to the nature of setting up and managing a private server it has to be noted that technical know-how is necessary where as other cloud services do not require that.	
<b>How does it work?</b>	
Install method: First, Owncloud needs to be installed and configured on a server <sup>43</sup> . After that clients can install programs to a pc locally via an installer, but it can also be used through their website without the need to download.	
How to sync a file: Owncloud allows users fine-grained control over what files they want to sync. Folders can be chosen by the users and exceptions for certain files or extensions can be made.	
How to retrieve a file: Files can be downloaded from the website, but will also be synced into the selected folder.	
How to share a file:	

<sup>42</sup> ownCloud Pricing - <https://owncloud.com/pricing/>

<sup>43</sup> For more information on the install process: [https://doc.owncloud.com/server/admin\\_manual/installation/](https://doc.owncloud.com/server/admin_manual/installation/)

<p>A file can be shared both from the installed client and the website. To share a file, a shareable link is created. The recipient is able to view the file on the website and download it from there without having an account.</p> <p>Possibility to sync extra folders: Possible and easy to do</p>
<p><b>Privacy</b></p> <p>Privacy seems well taken care off by OwnCloud as the users hosts' own server and therefore remains in control of all of its data at all times. There is the issue of adversaries, OwnCloud can be setup as a public server that is exposed to the internet and therefore can be targeted by adversaries one way to count this is to keep the server private a local network, this does however forfeit the flexibility of remote access without a VPN to that network.</p>
<p><b>Security</b></p> <p>Encryption</p> <p>Transit: TLS encryption</p> <p>End-to-end: yes</p> <p>Storage: yes</p> <p>Authentication:</p> <p>2 factor: yes</p> <p>Type: Name and password</p> <p>Tracking: No tracking whatsoever</p>
<p><b>Reminders</b></p> <p>The USP of owncloud is of course managing and hosting your own private server. This does however required owning hardware and maintaining software. A lot more effort is required to use this setup as opposed to cloud based solutions.</p>
<p><b>Conclusion</b></p> <p>In conclusion, OwnCloud offers an interesting value proposition for the more tech-savvy user because some knowledge is necessary for the setup of the private server. After setup OwnCloud is a privacy friendly service that allows users to maintain ownership of their own data.</p>

*NAS (Synology).*

<p><b>Introduction</b></p> <p>A NAS (network attached storage device) is a device with storage that is accessible through LAN. A company specialized in selling the devices is Synology<sup>44</sup>. We look</p>
---

<sup>44</sup> <https://www.synology.com/nl-nl>

<p>at Synology because they offer a complete package with application and a website. Synology is relatively easy to set up and use. It fits within the self-hosted client-server model.</p>
<p><b>Business model</b></p> <p>Synology's business model is simple: they sell hardware and make a profit margin on that. However, they do also sell licenses for additional services such as surveillance station<sup>45</sup> which allows users to use their NAS as a security camera recording device as well.</p>
<p><b>Functionality</b></p> <p>Backup:            yes  Synchronization:   yes  Sharing:            yes (if the NAS is made accessible from outside the LAN)  Versioning:         yes  Web interface:      yes  Applications:       yes  Offline use:         yes</p>
<p><b>User experience</b></p> <p>The combination of a web interface and offline programs make that a Synology NAS offers a good user experience that is easy to use. However, installing and configuring a Synology NAS requires more effort and knowledge which probably is too much to ask from everyone.</p>
<p><b>How does it work?</b></p> <p>Install method:  Buy the NAS, add HDD's in it, configure the NAS itself and configure the network to not block ports required by the NAS. After this initial install the NAS can be used through their web interface and installable programs.</p> <p>How to back up a file:  Installing the Synology backup program and configure it correctly. After that back-ups are done continuously.</p> <p>How to sync a file:  Syncing is done with a program in which folders can be selected to be synced.</p> <p>How to retrieve a file:  Files can be downloaded from the web interface, but will also be synced into the selected folder.</p> <p>How to share a file:</p>

<sup>45</sup> <https://www.synology.com/nl-nl/surveillance>

<p>A file can be shared through the web interface, this does however require the NAS to be accessible from outside the LAN.</p> <p>Possibility to sync extra folders: Possible and easy to do</p>										
<p><b>Privacy</b></p> <p>Offers anonymity: sort off, if used in a closed LAN situation yes. Privacy, assuming the security is in order, is completely up to the users themselves. By hosting their own NAS they have complete control over their data. No one else can access it, which makes for an ideal scenario to guarantee privacy.</p>										
<p><b>Security</b></p> <p>For security there are two scenarios: NAS accessible over the internet This relies on the fact that Synology has secured access to the NAS in a good way. However there have been security breaches in the past [20]. NAS only accessible over LAN This is the safer option of the two as the security now is managed by the NAS itself but also by the firewall, which in principle blocks all external access to the NAS.</p> <p>Encryption</p> <table> <tr> <td>Transit:</td> <td>SSL/TLS encryption</td> </tr> <tr> <td>End-to-end:</td> <td>no</td> </tr> <tr> <td>Storage:</td> <td>yes</td> </tr> </table> <p>Authentication:</p> <table> <tr> <td>2 factor:</td> <td>yes</td> </tr> <tr> <td>Type:</td> <td>Name and password</td> </tr> </table> <p>Tracking: No tracking whatsoever</p>	Transit:	SSL/TLS encryption	End-to-end:	no	Storage:	yes	2 factor:	yes	Type:	Name and password
Transit:	SSL/TLS encryption									
End-to-end:	no									
Storage:	yes									
2 factor:	yes									
Type:	Name and password									
<p><b>Reminders</b></p> <p>Backup integration with regular cloud storage providers</p>										
<p><b>Conclusion</b></p> <p>A Synology NAS offers a good solution to users with a bit more technical knowledge. Setup and configuration are the most difficult, after that the user experience is good and the system is easy to use. For our custom solution there are few lessons to be learned.</p>										

## 9.5 Conclusion

In this section we saw several services that use different models to achieve data backup, synchronization and sharing. From the existing service we can conclude that each model offers a unique user experience.

Cloud based solutions (client-server model) offers the best user experience. It is as simple as installing a program with some minor configuration and it is ready to use. The downside with cloud based solution is privacy. It is hard to guarantee that privacy is respected, so trust in the companies behind the service is required.

Peer-to-peer based solutions take away this privacy concern. They do this by being a distributed system that is not owned by a company, but every user contributes to the system. To achieve this however some positive user experience is traded in. For example if a user makes a p2p system with two nodes and one node goes down, file sync will not be possible. So users need to be more aware of their data and on which nodes it is saved.

Self-hosted solutions can use either abovementioned model. The obvious example would be client-server model by using a NAS or self-hosted server. This gives the user control over privacy and security measures of its data and allows them to create a user experience that fits their specific situation. However a lot of resources in the form technical know-how to setup and maintain the server and financial resources to own or rent the hardware required are the downside of the self-hosted model.

If we look back at what problem we have, anonymous file sharing. This eliminates cloud based solutions and therefore the client-server model. Next up, we can eliminate self-hosted solutions as the upfront investment cost, effort of maintaining and technical know-how required do not fit the requirement of easy to use. So by using decision making by elimination we can conclude that p2p offers the best solution for our solution.

Within peer-to-peer, Resilio Sync seems a promising solutions to investigate. It does lack receiver anonymity but we expect that this can be added to it.



## 10 Appendix II: Token typing with encrypted strings in partially blinded signatures

In this appendix we look at an alternative solution to typing tokens with different key pairs. Instead of using those different keypairs we only use one and include the type of the token within the token itself by using partially blinded signatures. To do this we rewrote the text from section 6.2 to suit and explain this alternative solution below.

### Token incentivizing system in short.

In this section we will describe the complete workflow of the new token system. For this workflow we assume a running environment with several torrent sessions. We will talk about initializing the system in a later section. For this explanation we use figure 1 on the next page. When for example (3) is used in the upcoming text we mean figure 1 sequence number 3.

Our starting point is a peer that wants to join a torrent session. Therefore, the peer joins a blender and makes itself available for relaying by sending a join request (2). To successfully join the blender the peer has to send a blinded self-generated serial number (1) to the blender. Why this blinded signature is needed will be explained later. Then the peer must wait for the first blender request.

Upon receiving a peer request from the tracker (3), the blender sends its first request to all peers that are in the blender set, it calculates the probability  $p$  as in the normal BitBlender protocol and for each peer it also signs their blinded serial number (4). The blender sends the signed blinded serial number back to the peer together with a cipher text that the blender created using authenticated encryption (5). That ciphertext contains the `info_hash` of the torrent session to join and a nonce to create unique ciphertexts. This ciphertext assures that the peer cannot use the token for a different session than it is supposed to. The combination of the signed serial number (blinded or not) and the cipher containing the `info_hash` and we call a token.

The peer now decides, according the normal BitBlender decision process if it should join the torrent session or not (6). In any case, it creates a new blinded serial number (7) and sends that to the blender to indicate that it is ready for a new join request (8). If the peer also decides to join, the peer unblinds the signed serial number (9) and sends that combined with the cipher it got from the blender and a newly created blinded serial number (7) to the tracker to indicate that it wants to join the torrent session (10).

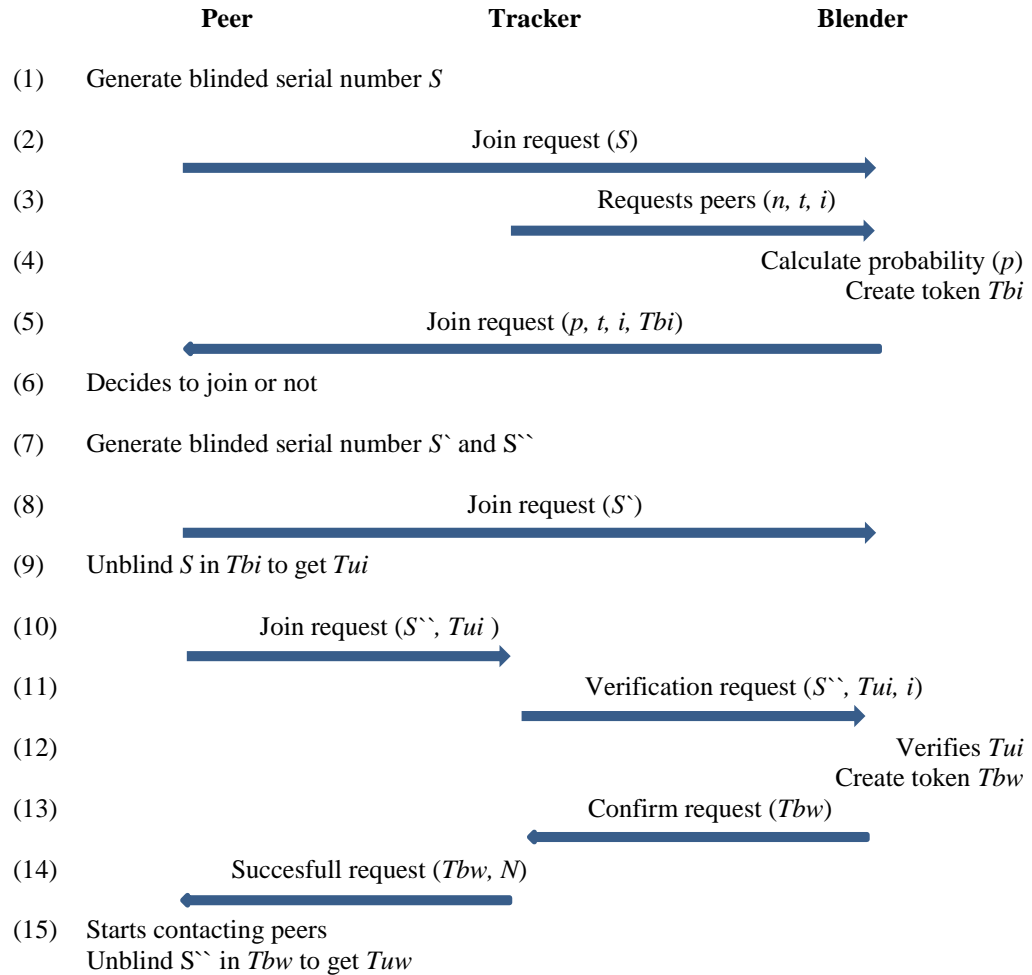
The tracker blindly forwards the token, new blinded serial number and the `info_hash` of the torrent session to the blender (11) and awaits the blenders response. At this stage the tracker cannot learn if the peer is a relay or not.

The blender verifies the signature on the serial number part and if it is correct, decrypts the cipher accompanying it (12). If that cipher contains an `info_hash` that matches the one sent directly by the tracker, the blender sends an approval message to the tracker (13). Accompanying this approval message, the blender also creates a new token (12). This token contains the signed blinded serial number and the cipher. This time the cipher does not contain a `info_hash` but a predefined string (with equal length as an `info_hash`) that represents a wildcard which allows the peer to join a random session.

There is an exception to this wildcard token in the form of an empty token. If a peer joins the session with a wildcard token, it should not receive a new wildcard token as reward. To prevent the tracker to be able to identify peers by seeing who gets a token in return and who does not an empty token is created. Similar to the wildcard token but this time a predefined string that represents a worthless token. The peer itself knows as what type it joins the session and can discard this empty token upon receipt.

The tracker receives the approval with the new token from the blender and sends the new token with its normal BitTorrent response to the peer (14) who can then join the torrent session and unblind the serial number in his newly received token (15) to get a usable wildcard token.

Fig. 1. Token incentivizing protocol overview



$S$  = Serial number

$n$  = Amount of relay peers

$t$  = Tracker URL

$Tbi$  = Token consisting of blinded  $S$  and authenticated encrypted info\_hash

$Tui$  = Token consisting of unblinded  $S$  and authenticated encrypted info\_hash

$Tbw$  = Token consisting of blinded  $S^{\prime\prime}$  and authenticated encrypted wildcard string

$Tuw$  = Token consisting of unblinded  $S^{\prime\prime}$  and authenticated encrypted wildcard string

$N$  = normal response of BitTorrent containing peer list.

**Peer.**

A peer has got some additional responsibilities with the new protocol. If it wants to join a blender it will have to create a unique serial number  $S$ , blind it and send that to the blender with the join request.

When the blender sends request to join a torrent session the peer will receive the following request;  $(t, i, p, T)$ .  $t$ ,  $i$  and  $p$  stand for tracker identifier, info\_hash and probability respectively. The  $T$  is the token that entitles the peer to join that specific torrent session as a relay.

To join a torrent session as a relay peer, the peer now must include the token  $T$  in the request and a new blinded serial number to such that the blender can create a new token based on it. However, before sending this request, the peer needs to unblind the serial number part of the token. This blinded serial number will be explained later in this section.

The response of the tracker can be an error state (torrent session is not joinable) or a successful state. In the case of successful state, a new token is sent with it that can be used as a wildcard token for any random torrent session. Allowing the peer to download a torrent without relaying.

When trading in this wildcard token the process is identical as described for a torrent specific token. Except for the fact that the tracker will now return an empty token which can be discarded. To join another session the peer has to restart the process and join a blender.

Peers cannot see the content of the token, therefore the peer itself is responsible of keeping track of the type of the token. So, whether it is a wildcard, torrent specific or empty tokens. Empty tokens can be discarded upon reception as they hold no value.

**Blender.**

The blender undergoes the biggest modifications as it will now be responsible for token creation and verification. We lay the verification of the token with the blender to reduce overhead for the tracker and to keep all cryptographic actions in one place. To following addition to the blender have been made.

Instead of only calculating the probability  $p$  and sending that to all relay peers that are in the set of the blender called  $B$ , it now must generate a token for each  $b_i$  in  $B$ . Let a torrent specific token be  $T$ . For each  $b_i$  in  $B$  a token  $T$  is created. The blender has a secret symmetrical key  $k$  that it uses for encryption of token creation and verification. To each  $b_i$  in  $B$  the blender sends the following data with the request  $(p, t, i, T)$ . Where  $p$  is the probability,  $t$  is the tracker url,  $i$  is the info\_hash. So,  $T$  is added to the original request.

*Token creation.*

A token consists of two parts; the serial number and the torrent identifier.

For token creation the blender uses the following values;  $bS$  for the blinded serial number and  $I$  which contains either the info\_hash or the predefined string for a wildcard or empty token.  $bS$  is obtained from the peer whereas  $I$  is decided by the blender itself depended on the type of token it wants to create.

It is important that the tracker cannot see the content of  $I$  as that would leak information about what type of peer it is. Therefore,  $I$  is encrypted with an authenticated encryption method using a secret key  $k$ . After that  $S$  and encrypted  $I$  are signed with the blender private key  $sk$  to create a token. So, a token  $T$  is created as followed:

$$T = \text{Sign}((bS, AEnc(I, k)), sk)$$

*Token verification.*

Verification requests come from the tracker and have the following form:  $(i, T, s)$ .  $i$  is the info\_hash,  $T$  is the token and  $S$  is the new blinded serial number. Note that the token does not contain  $bS$  anymore by  $S$  as the peer has unblinded the serial number.

Verification of the signature happens with the blenders public key  $pk$ .

Verification and decryption happen as follows.

$$S, I = \text{Verify}((S, ADec(I, k)), pk)$$

After the signature has been verified the blender has  $S$  and  $I$ . The blender starts by checking if  $S$  is a unique serial number that the blender has not seen yet. If that is the case it continues with verifying  $I$ , if  $S$  is already invalidate the blender returns an error state.

For  $I$  it checks its contents to see if it is one of the predefined strings or possibly an info\_hash. If it is the predefined empty string, the blender returns an error to the tracker. If it is a predefined wildcard string, the blender responds with a successful state, combined with a new empty token which is explained at the end of this paragraph. If  $I$  contains an info\_hash it is compared with the info\_hash  $i$  that the tracker sent. If they match, the blender responds with a successful state and a new wildcard token.

For successful states the blender invalidates  $S$  such that it cannot be used again after this transaction. For each successful state a new token must be created as mentioned before. The new tokens that are sent with successful states are based upon the  $s$  that is sent by the tracker.

### **Tracker.**

The tracker only has to facilitate the new authorization method, meaning it will have to receive the token and new serial number from the peer and send it in a request with the info\_hash of the corresponding session to the blender for authorization. If the blender responds positively to the request the peers request gets answered in normal fashion with the addition of a new token for the peer. If the blender responds negatively, the request is denied. The tracker does not need to know or care about the type of token therefore protecting the anonymity of the peer.

### **Preventing double spending with partial blind signatures.**

To prevent double spending of tokens we use blind signatures as introduced by David Chaum [5] and used in e-cash protocols. Blind signatures work as follows:

The requesting party creates a message, blinds the message and send that blinded message to the signer. Blinding can be done by combining the message with a blinding factor. The signer signs the blinded message using a normal signing scheme and returns it to the requestor. The requestor can then unblind the message whilst keeping the signature valid when using for example RSA [5]. We do note, however, that RSA is not perfect and further research is required to choose a suitable signing method [4]. We will use blind signatures as an interactive protocol, not an actual implementation.

The implementation in our token system is as follows: the peer makes a unique identifier, a sort of serial number as used in cash money. This serial number will be used to prevent double spending, meaning the blender will invalidate it upon spending. To validate this serial number the peer first blinds it serial number then sends it to the blender. The blender verifies if the peer is entitled to that signature, signs it and sends it back to the peer. The peer unblinds the serial number uses it when joining a torrent session. The blender is responsible of validating that serial number, it does this by verifying that it has not seen this serial number before and invalidates it afterwards to assure it is not usable again.

This leaves us with one problem. The blender cannot see the serial number but does need to make sure it is only entitled to a specific torrent. To do this we use partial blind signatures [1, 21]. The basic principle is the same, but now the blender adds a string to the end of the blinded part. The peer only has to unblind the first string and leaves the second string unchanged. More research is needed to confirm if this works as described.

Why go to all the trouble of using partial blind signatures? The answer is to defend against a malicious blender. If we would let the blender create and distribute the serial numbers it can simply remember to what peer it send it. If it sees that serial number again when validating it, it knows what peer it belongs to.