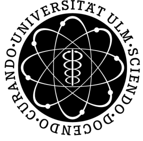

ulm university universität **u**ulm

Ulm University | 89069 Ulm | Germany

Faculty of Engineering,
Computer Science,
and Psychology
Institute of Databases
and Information Systems

Tinzenite: Encrypted Peer to Peer File Synchronization via the Tox Protocol

Master thesis at Ulm University

Author:
Tamino P.S.M. Hartmann
tamino.hartmann@uni-ulm.de

Reviewers:
Professor Doctor Manfred Reichert
Professor Doctor Martin Theobald

Supervisor:
Marc Schickler

Year:
2015

Version October 28, 2015

© 2015 Tamino P.S.M. Hartmann

This work is licensed under the Creative Commons. Attribution-NonCommercial-ShareAlike 3.0 License. To view a copy of this license, visit
http://creativecommons.org/licenses/by-nc-sa/3.0/de/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Setting: PDF-LaTeX 2$_\varepsilon$

Abstract

We proposed and implemented an open source, peer to peer, file synchronization software based on the Tox communication protocol. Targeted features include full secure communication between peers, an encrypted server peer, and a focus on ease of use while retaining data security. The software suite was implemented based on the Tox protocol, with Golang as the programming language, and the server client built to offer free choice of storage mechanisms, for which we implemented support for the Hadoop distributed file system. The proof of concept implementation was shown to work and further possible work discussed.

Gratitude

My gratitude goes out to my support network from my university, my family, and my friends. Especially my supervisor Marc Schickler for offering encouragement and praise when things worked – I appreciate the support. He also offered up excellent points for improving my thesis in his corrections. My extended family for the many nice questions about what I was doing and sitting through the long explanations with sufficient interest to keep me going. Maybe someday I will find a single sentence that sufficiently explains everything comprehensibly for "non-computer" people. My aunt Sibille and my parents Katja and Stefan for their hard work of proof reading this thesis: all remaining errors are my own. My friend Andreas for providing the affidavit translation and finding one error. Furthermore my friends for regularly asking how I was doing, thus building up the pressure to keep working continuously.

This work would not have been possible without the great tools and libraries and the many people that implemented and documented them. Therefore my heartfelt thanks go out first and foremost to the open source community that made building this thesis a possibility: Tox for providing a peer to peer fully encrypted communication channel and Golang for the access to source code when things didn't work as expected. Worthy of special mention is the Github user Codedust for maintaining the Golang Tox wrapper on which I built this thesis. Special thanks are due for their support whenever I had problems or feature requests and always received help and support.

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Goals 3

1.3 Name 4

1.4 Structure 5

2 Related and Existing Work **7**

2.1 Existing Software 7

2.1.1 Client-Server Solutions 8

2.1.2 Peer to Peer Solutions 9

2.1.3 Additional Security Layers 11

2.1.4 Comparison 12

2.2 Related Research 12

2.2.1 What is a File Synchronizer? 12

2.2.2 An Algebraic Approach to File Synchronization 13

2.2.3 Peer-to-Peer Reconciliation Based Replication for Mobile Computers 14

2.2.4 Perspectives on Optimistically Replicated, Peer-to-Peer Filing . . . 14

2.3 Conclusion 15

3 Concept **17**

3.1 Basic Goals 17

3.2 Features 18

3.2.1 Scope of Work 19

3.3 Dependencies 21

3.3.1 Golang 21

3.3.2 Tox 22

3.3.3 Hadoop File System 23

3.4 Software Scope 23

3.4.1 Tinzenite Core Library 24

3.4.2 Client Peer 24
3.4.3 Server Peer 25
3.4.4 Web Interface Peer 25
3.4.5 Mobile Peer 26
3.4.6 Passive Peer 26

4 Architecture 27

4.1 Meta Data 27
4.1.1 Organizational Directory 29
4.1.2 Removed Directory 31
4.1.3 Unsynchronized Directories 31
4.2 Object Model 32
4.2.1 Directory Model 34
4.2.2 File Model 35
4.2.3 Object Operations 36
4.3 Communication Specification 41
4.3.1 Connection Management 41
4.3.2 New Connection Establishment 43
4.3.3 Message Order 45
4.3.4 Trusted Synchronization 46
4.3.5 Encrypted Synchronization 49
4.4 Update Detection and Reconciliation 51
4.4.1 Update Detection 52
4.4.2 Update Reconciliation 52
4.5 Security Considerations 55

5 Implementation 57

5.1 Tools and Environment 57
5.1.1 Golang 57
5.1.2 JSON 59
5.1.3 Tox Binding 59
5.1.4 Hadoop Client Binding 60

5.1.5 Environment 60
5.2 Software Structure 60
5.3 Highlights 62
5.3.1 Model 62
5.3.2 Channel 64
5.3.3 Tin Program 64
5.3.4 Bootstrap 65
5.3.5 Server Program 66
5.3.6 Golang Issues 68
5.4 Security 70
5.4.1 File Encryption 70
5.4.2 Key Encryption 71
5.4.3 Challenge Response 73

6 Results and Recapitulation **75**
6.1 Comparison 75
6.1.1 Network Performance 75
6.1.2 Usability 76
6.1.3 Security 77
6.2 Future Work 78
6.2.1 Improvements to Existing Work 78
6.2.2 Expanding Work 82

7 Conclusion **87**
7.1 Theoretical Work 87
7.2 Implementation Work 88
7.3 Closing Statement 88

8 Glossary **89**

1 Introduction

The widespread adoption of the internet in modern culture has caused a large evolution in the way we use computers. When computers were still new almost all programs and services were run locally on the machine. Nowadays our usage has shifted to a multitude of online services and solutions. One of the more well known examples is digital encyclopedias: Microsoft Encarta was distributed on a single CD upon release. But today Wikipedia has become synonymous with looking up information – a large encyclopedia service which can be accessed via the internet. Other examples can be found a plenty: almost anything is available as a cloud service, from games to web hosting.

Therefore it should come as no surprise that storing data has also increasingly shown trends to moving onto the internet. And indeed the multitude of online data storage services in existence today show the rising trend to store data – not on personal machines – but to entrust it to third party service providers. Such third party services provide users with access to their data wherever they can access the internet in general. Dropbox [Dro] made its name as a popular data storage service provider, but existing internet giants were quick to follow. Thus both Google and Microsoft also began offering competing services: Google Drive [Goo] and OneDrive [One] respectively.

All was well for a while. Users merrily entrusted their data to third parties under the reasoning of ease of access and ease of use. Private data would no longer be lost if the personal computer stopped functioning for one reason or the other. Additionally the rise of mobile devices such as smartphones only quickened the adoption of cloud storage services. The exorbitant pricing of storage space and the lack of removable external

storage on these mobile devices drove people to solutions that allowed them to access all their data nonetheless.

In 2013 Edward Snowden revealed that the blind trust in these third party services was misplaced. Thanks to Snowden's whistle blowing and personal sacrifice a global conspiracy of governmental agencies that undertake massive online surveillance of most internet communications was brought to the public attention. Since the majority of the global internet players are based in the United States of America, this had huge implications for all services offered via the internet. Many services used by the majority of internet users have been shown to be either at risk of being or already are compromised or even known to explicitly include capabilities for compromising data security. This includes internet giants such as Microsoft, Yahoo, Google, Facebook, Paltalk, AOL, Skype, YouTube, and Apple [GP13]. Snowden also revealed that most data passing through internet exchange points is surveilled [Hol14]. These form the backbone of the internet in its current state.

While public outrage lasted just as long as the major news outlets covered it, the revelations had a major impact in the technical community. New software solutions were required to enable users to reconquer their privacy on the internet without sacrificing usability and ease of access. One example is the Open Whisper Systems group. Their stated goal on the website is "[...] we're working to advance the state of the art for secure communication, while simultaneously making it easy for everyone to use." [Whi]. Another example is the Tox instant messenger community that is building a free, open source, peer to peer Skype alternative [Toxa].

1.1 Motivation

The thesis is mainly motivated by the revelations of drag net surveillance by Edward Snowden and the compliance of so called trusted service providers within their legal obligations and even beyond. The uncovering of the global surveillance network has shown that we can not entrust third parties with our data without taking additional steps to secure it. While services such as Boxcryptor [Box] exist to add a layer of encryption

on top of the data storage services they bring with them a few disadvantages in terms of ease of use. Notably for this specific example access to the encrypted data is severely hampered because it can not be accessed via the internet without first decrypting it additionally.

Services exist that do not rely on a user's trust to a third party. These peer to peer solution promise to keep all data of the user only on the user's machines. However the lack of a copy of the data on a remote server removes the advantages of web access over the internet and decrease their utility as data backup services. Examples for existing peer to peer solutions include BitTorrent Sync [Bitb] and Syncthing [Syn].

Our motivation stems from the hope that we can combine the two types of offering data storage services into one service that retains most of the advantages. Thus we propose the implementation of a peer to peer, fully encrypted file synchronization system that is capable of storing encrypted backups of users' data on third party servers while still allowing remote access to it via the internet. We therefore define two types of peers: trusted peers that store data unencrypted and are intended for the trusted devices of users and encrypted peers that remove the need to trust third parties while still allowing them to serve as remote storage providers.

1.2 Goals

This work thus has the goal of designing and implementing a proof of concept that such a service is possible. The proposed implementation should build on peer to peer communication, bypassing the requirement of a centrally hosted third party service. Unlike most existing peer to peer solutions we propose to include support for encrypted third parties so that the advantages of remote storage are kept. To harden such a peer to peer network it is necessary to encrypt all communication between the peers. Instead of integrating and mixing encrypted communications with the file synchronization commands required for such software to work, we utilize an existing secure peer to peer communication infrastructure in the form of Tox, on top of which we will propose a protocol for the sole task of file synchronization. The encryption scheme is to be chosen

so that while encrypted peers are denied access to a user's data, authorized access over the internet is still possible. To facilitate a high flexibility of the encrypted peer we will also work to implement a storage interface that allows any desired storage system to be used with a user's data, from direct disk storage to distributed file systems.

The implementation of the program will be done with Golang, building on Tox for the communication and the Hadoop distributed file system for storing data on the encrypted peer. We will expand on the specification and define the scope of this work in preparation. Apart from the theoretical work a large part of this thesis is also the implementation of a proof of concept. Therefore we will also discuss problems we encountered while working on the software aspect and our solutions to them. Finally we will look back at the implementation and compare it to the theoretical ground work. For a better comparison a brief discussion on similarities and differences to existing file synchronization services will also be included.

1.3 Name

Figure 1.1: The preliminary icon of our implementation of Tinzenite.

To differentiate our proof of concept implementation from existing solutions a unique name was required. We chose the rock-forming mineral tinzenite for the name. We wanted a name that had some association to a crystal to signify the hardened security aspect of our work. Furthermore the crystal has a unique orange color that lends itself well as an icon as can be seen in figure 1.1. Specifically, Tinzenite is the name of the peer network which in turn is compromised of two distinct software peers which are based on a common protocol and communication standard.

1.4 Structure

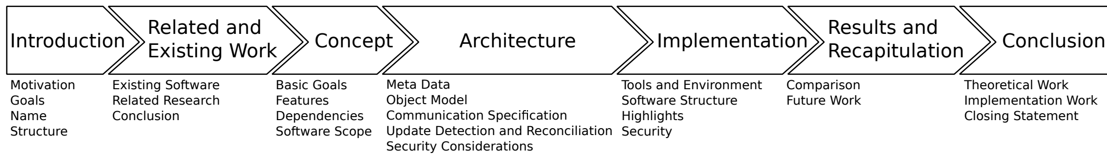

Figure 1.2: A graphical representation of the structure of this thesis.

Chapter 2 presents existing software that is relevant to our work, as well as related papers. Built on this we will define the concept of this thesis in chapter 3. Chapter 4 describes the theoretical architecture that Tinzenite will be based on. The implementation of the architecture into our proof of concept implementation will be discussed in chapter 5. We will expand on the completed work in chapter 6 and note what possible future work could be built on top of the provided thesis. Finally chapter 7 will conclude this thesis and provide a general closing statement. Figure 1.2 shows a graphical representation of the structure of this thesis.

2

Related and Existing Work

This chapter serves two purposes. First we will discuss existing solutions. The differences between these and our proposed work will serve to highlight in what ways our work expands existing solutions. Then we will take a look at the academic side of file synchronization and discuss related papers. These will serve as a foundation for how we implement the Tinzenite protocol.

2.1 Existing Software

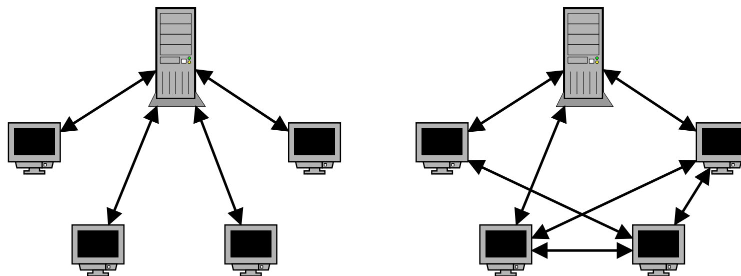

Figure 2.1: Example diagrams of a server client architecture (left) and a peer to peer architecture (right). Note that the peer to peer architecture does not exclude servers as peers.

For any internet technology there are two options for how to structure its architecture in general. Most of the internet today is cleanly divided in a client-server structure where a client always requests information from a central server. This is a strongly hierarchical structure. An example of this is downloading a PDF from a website. The other option is a distributed peer to peer model, where a client requests information from any other client and in turn will also respond to queries from other clients. One of the more well

known candidates of this is the Bittorrent protocol [Bita]. Both options have been used in existing file synchronization solutions. See figure 2.1 for an example diagram of the two architectures. we have divided the existing software between these two extremes and will shortly discuss these in the following sections.

2.1.1 Client-Server Solutions

For client-server solutions any user must rely on the availability of the central server. These are often hosted by third parties in a distributed manner for reasons of performance and scaling. Providing a central service that is required for the file synchronization to work also allows easy monetization. Since a large amount of such services exist, we will highlight only the three most popular ones for this thesis. These are respectively: Dropbox with 300 million users, Microsoft's OneDrive with 250 million users, and Google Drive with 240 million users as of 2014 [Gri14].

Dropbox

Dropbox [Dro] is one of the most popular cloud storage providers currently and is known by many users looking for a solution that works across multiple operating systems. Positive features include web access to stored data, clients for many different platforms, easy sharing of data with outsiders, and ease of use. On the negative side the service relies on its back-end servers, although computers can synchronize files between themselves on a local area network. Dropbox also lacks any end to end encryption, but does encrypt data while in transit and when "at rest" in their data centers [Bor14]. Therefore it comes with little surprise that they were prominently featured in the Snowden leaks [RT 14]. The company does have full access to any data the user uploads to their servers, as long as the users do not encrypt the files beforehand themselves. Therefore we can extrapolate that Dropbox can offer up a user's data if required to by a fourth party, as is the case in the United States of America via PRISM by the NSA [GP13].

Dropbox offers a free account for a basic storage plan. Additional storage capacity is available for purchase or by referrals that lead to account creations. None of the core

applications have been open sourced to date, meaning even if Dropbox implemented strong encryption it could not be independently verified. This includes client software and server software.

OneDrive

With the release of Microsoft's newest operating system the company has increased its push for public adoption of its OneDrive [One] cloud storage service. It is now directly integrated within Windows 10 and is opt-out for users that do not wish to utilize it. OneDrive is integrated with Microsoft's answer to Google Docs, Office 365. Furthermore more free storage is added for registering devices and using Microsoft's products.

Similar to its two competitors, the base plan is free, with the option of paying monthly for an increase in storage space. No software for OneDrive has been open sourced.

Google Drive

Google Drive [Goo] is similar to Dropbox in term of its functionality. It does go a step further than Dropbox by integrating tightly with their suite of online applications for creating and editing documents, named Google Docs. Google also has full access to all data that users upload to their servers. This in turn means that under the PRISM program by the NSA, all such data is also retrievable by a fourth party [RT 13].

Google also offers a basic storage amount at no charge for the user. Again additional space can be purchased on a monthly basis. Apart from offering access to fewer clients than Dropbox, Google also has failed to open source the components of its service.

2.1.2 Peer to Peer Solutions

Peer to peer systems work without a central server. The trade off is that such solutions require assistance in locating other peers, which means that some form of a peer discovery system must be implemented. While this could be done via a central server,

nowadays the common solution to the problem is a distributed hash table [RFH+01] which is used to look up the required peers. Once another peer has been found the connection can be established. Apart from not relying on the availability and trustworthiness of a central server, peer to peer solutions offer a possible performance advantage as they can distribute bandwidth between every peer, even actively unchoke a saturated peer. As an added bonus the path data takes between two peers will always be the most direct path as no data must first go towards a third party. This bonus can be lost if onion routing is implemented for anonymity purposes.

BitTorrent Sync

An existing solution for a peer to peer data synchronization service is Sync [Bitb] from the BitTorrent company. It is built on existing BitTorrent technology and thus keeps many of the positive features associated with BitTorrents: reducing the impact of transferring large files over a limited network. Therefore it has no central point of failure and can even run without internet access by transferring files between computers on a local area network directly. However, just as the client-server solutions before, Sync is closed source.

Clients exist for a multitude of platforms. While information is encrypted in transfer, there is no differentiation for untrusted clients. Sync is free for basic usage, but requires a paid account for additional features that offer more fine grained control. This includes advanced data sharing options. Unlike Syncthing BitTorrent Sync is capable of sharing content to third parties that are not part of the data storage service.

Syncthing

Syncthing [Syn] is an open source file synchronization software on an equally free block exchange protocol. This protocol is a mixed data transfer and communication protocol in one, with encryption for all communication built in directly. Again the user can not designate untrusted peers or specify peers to retain only an encrypted copy of the data.

Unlike the other solutions mentioned here, Syncthing is open source and can thus be independently verified to be secure. Like Tinzenite it is also implemented in Golang. The downside is that Syncthing lacks a feature which BitTorrent Sync shares with most client-server solutions: the capability of sharing a specified file or directory with other users who are not part of the storage system.

2.1.3 Additional Security Layers

It is of course possible to encrypt all data beforehand before using any data storage service such as those mentioned. This has a few notable downsides however. For many client-server solutions the user loses out on web access to the data along with a host of additional feature built on the availability of the data. While advanced users may be capable of encrypting their data themselves using a range of available tools, such solutions require the user to manage the encryption keys themselves. This is a non trivial issue beyond adding an additional hurdle to using data storage services. As so often commercial solutions for this exist as the following example shows.

Boxcryptor

All three server-client services mentioned are not to be trusted with private information that must remain secure due to possible fourth party access. However solutions for encrypting the data sent over such services which manage the encryption keys exist. One example of such a service is the Boxcryptor software [Box]. It encrypts all data before uploading it to the connected cloud storage service and decrypts it when retrieving it. The user keys are attached to the user's account. Asymmetric encryption is used to upload all keys to the companies key server so that other clients can retrieve and, with a correct password, decrypt the data. Sharing of data is still possible even for users without an account by utilizing special keys.

The downside to this approach is mainly that it has to be used on top of an existing cloud storage service. This effectively means that the user must run an additional program on top of the file storage service to ensure secure data. Boxcryptor can be used with most

existing third party data storage systems. Boxcryptor offers a free version of its service, but to access all security features a paid subscription is required. The software is not open source and thus not independently verifiable.

2.1.4 Comparison

In this section we will briefly highlight the capabilities and differences between the discussed existing solutions. We chose to compare the features that we believe to be most important to this thesis. See table 2.1.

Name	Architecture	Free Storage	Client-side Encryption	Open Source	Account Required
Dropbox	Centralized	2 GB			✓
OneDrive	Centralized	15 GB			✓
Google Drive	Centralized	15 GB			✓
Boxcryptor	–[a]	–[b]	✓		✓
BitTorrent Sync	Distributed	∞			✓
Syncthing	Distributed	∞		✓	

[a] Boxcryptor is used on top of an existing cloud storage provider. The encryption infrastructure itself is centralized.
[b] Boxcryptor is free to use but many features are only available for a paid account.

Table 2.1: This table serves to highlight the differences between the existing software previously discussed.

2.2 Related Research

The following discussion concerns papers that we believe have an impact on our work. We discuss their general content with a focus on what information we extracted and how we believe we can apply it to Tinzenite.

2.2.1 What is a File Synchronizer?

The paper by Balasubramaniam and Pierce [BP98] has the stated goal of offering a framework for describing the behavior of file synchronizers. Notably the paper's authors

divide the process into two separate phases: update detection and reconciliation. Update detection is defined as the recognition of where updates have been made to the directory since the last synchronization. Reconciliation is defined as the combination of updates between peers to transfer all peers to a new, synchronized state.

For Tinzenite we assimilated a few things out of this paper: primarily the distinction between the update detection and the reconciliation phase. We will also initially ignore links and file permissions due to increased complexity, but unlike them we will never modify a file ourself. The paper also gave us a definition for the type of update detection strategy we wanted to try: namely the modtime inode strategy. Ideally we would take an on-line update detector but this seems impossible to do under Linux currently for an unlimited amount of files. Also of interest is the differentiation between pull and push models for propagating updates. Tinzenite intends to be a hybrid between the two: while the main work will be done via push – once peers have established a connection – updates may also be propagated via pull for performance reasons. In the reconciliation section of the paper is the listing of the various possible states that two replicas of a directory can possibly reach given a set of operations, which is also of interest for this thesis. We intend to ensure that the core protocol for Tinzenite will be capable of handling all of them to the best of its abilities.

2.2.2 An Algebraic Approach to File Synchronization

The paper by Ramsey and Csirmaz [RC+01] presents an algebra for reasoning about file system operations with the intended purpose of specifying an algorithm for file synchronization. Interesting for our work is the discussion of possible commands that can be executed on said file algebra. While from a user's point of view a multitude of operations seems possible (create, remove, rename, move, derive, and edit) the paper's authors distill these down to just three for the technical side: create, remove, and edit. Rename can be executed by removing the original file and creating a new file with the new name while keeping the content identical. Move can be executed much the same but keeps the name, instead changing the location where the new file is created. Derive is argued to be indistinguishable from edit without higher level knowledge. While

not impossible to detect, this feature we consider to be beyond the scope of Tinzenite. Features that Tinzenite shares with the paper's prototype include that parents must be created before their children and that children must be removed before their parents.

The paper also offers a number of improvements that Tinzenite will or could incorporate. Notably this includes support for an explicit move command. This would remove the need for removing and recreating a file's content when it was moved, thus making this operation more performant.They also suggest using fingerprints of content to avoid sending data that already exists. Tinzenite already utilizes content hashes, so expanding it to be more intelligent about when to request actual data to be sent should be easily implementable by comparing hashes of files before requesting transfers.

2.2.3 Peer-to-Peer Reconciliation Based Replication for Mobile Computers

In [RPG+96], Reiher et al. nicely state the benefits and drawbacks of using a peer to peer system for file synchronization in regard to mobile devices. Benefits include not having to rely on an permanently available server, being able to transfer files without access to a central server, and transferring files over the shortest available network path. The authors list as drawbacks the higher required complexity of the algorithms used to control the replication. They conclude that peer to peer replication is well suited for mobile devices.

2.2.4 Perspectives on Optimistically Replicated, Peer-to-Peer Filing

The paper by Page et al. [PJGH+98] is notable in our case for two main reasons. It offers a nice set of definitions of problems that we must also solve for Tinzenite and even offers relevant solutions. The authors also discuss the performance of their implementation. Focus of the paper is the evaluation of the use of optimistic replica consistency, automatic update conflict detection and repair, the peer to peer interaction model, and the Ficus design and construction.

Relevant for Tinzenite is the paper's solution to the insert delete ambiguity. The solution is to keep a list of deletions until all peers know of the deletion. Only then can the deletion actually be garbage collected. In Tinzenite we will use a similar approach.

2.3 Conclusion

Based on the listed existing solutions and the briefly discussed related papers we have synthesized a set of features that we wish Tinzenite to have. These will be discussed in the next chapter. More importantly our research highlights a few problems that we will need to solve or use already existing solutions for. This starts with defining the operations we will allow on objects within a directory and continue on to how to solve ambiguities and merge conflicts when synchronizing between multiple peers.

3 Concept

The following chapter discusses all conceptual work that went into creating Tinzenite. We will first give an overview of the basic goals of the work created by this thesis. Next we will expand on the goals by discussing the features we would like Tinzenite to have and defining their scope. These will be based in part on the existing solutions discussed in the related work chapter. Based on the concept and the proposed features we will explain the software components we plan to implement.

3.1 Basic Goals

The stated goal of Tinzenite is to offer a peer to peer solution for file synchronization that builds on the strengths of Tox (see section 3.3.2). It is important to us to build the system in a way that it is secure from unauthorized access by third parties, even if they retain a copy of the data. In fact we propose an explicit client for third party support so that third parties can offer a storage peer as a service.

Therefore we will need to develop a protocol for a decentralized and distributed system that relies on the underlying secure channel. Based on this we hope to implement a proof of concept peer client for normal computers and a third party client that securely stores the user's data off site. This encrypted peer should allow the storage of data using the Hadoop file system (see section 3.3.3).

As proof of the protocol we will implement both programs in Golang (see section 5.1.1). The programs should cover the basic requirements for file synchronization and work correctly for all base cases. Thus we will have an actual implementation with which we

can compare our proposal against existing solutions and highlight derived problems, solutions, and differences.

3.2 Features

This section will define the features we would like Tinzenite to have, including features that go beyond the scope of this thesis. Therefore we will classify the features by scope. The exact features we would like to have have been synthesized from the existing and related work from the previous chapter.

1. **File Synchronization Protocol** Design of an extensible protocol on which all communication between peers will be based. The open specification will allow the development of compatible peers by others, important for the extensibility of the system.
2. **Peer to Peer Architecture** The complete software suite should run in a direct peer to peer mode to remove the requirement of third parties to facilitate data exchange and to remove the associated security risk. We also hope that this feature will allow clients to synchronize independently of the internet: peers should be capable of utilizing local connections directly.
3. **Secure Transport** All communication between all clients should always be fully end to end encrypted.
4. **Third Party Client** A dedicated client for untrusted third party servers that holds only an encrypted copy of the data.
5. **Shadow Files** It should be possible to avoid having to fetch unwanted files for space constrained clients. Dubbed shadow objects, this feature could be important for mobile devices as they run on power and bandwidth constraints.
6. **Delta Updates** Since it is wasteful to transfer redundant data when only small parts of files are changed, we would like to add the capability to only send the delta difference between two files.

7. **Object Atomicity** We will not touch the content of files, instead we will treat them as singular objects. This should help to guarantee that files are never modified by the system beyond the required operations for synchronization. In particular this forbids automatically merging changes in files. All conflicts must be resolved by the user: we do not intend to guess the correct resolution strategy for any file type.
8. **Passive Peer** Since the third party client already stores all data fully encrypted, support for a passive encrypted peer could be easily added. This would allow the user to use storage devices as additional peers which can be activated by pointing Tinzenite at them whenever they are connected. Much like using mobile active peers as data bridges this feature would allow passive peers to also serve as data bridges while keeping the data fully secure.
9. **Performance** The proposed protocol should allow for the client software to run as unobtrusively as possible. This includes requiring only the bare necessity of performance for all operations and avoiding redundant work.

Please note that many further features are not dependent on the capability of the Tinzenite system but on the implementation of peers. An example for this is web access to an encrypted storage: this is a feature that explicitly can be implemented in a secure way[1].

3.2.1 Scope of Work

In this brief section we will go through the exact scope that this work is to fulfill. We therefore divide the above features into three categories, ranging from those required to have for the thesis to be considered successful, to those that can be added as extras if time permits or as future work – see section 6.2. Furthermore we will expand on the actual implementation work we intend for each scope definition.

[1]The key for decrypting the data can be unlocked by users in the web browser by entering the correct pass phrase. Utilizing the shadow file capability the web application would act as a temporary trusted peer until the users are done with their data access.

MUST Have These features are required for the thesis to be considered basically successful. This means that the basic fundamentals of the proposed complete scope have been met and are in working order. Specifically this includes a fully working computer client based on a specified API on which all future work can be built on. This client must offer the basics required to get the system to work in a user friendly manner from setup through daily usage. Data transfer between multiple trusted clients must work as expected with collision detection and correct version iteration upon updates. Notably this will require of us to implement the entire communication stack and the directory management parts of Tinzenite.

File Synchronization Protocol The core protocol must be fully capable of basic file synchronization.

Peer to Peer Architecture The base client and library must be capable of running without a centralized system.

Secure Transport All communication must be fully encrypted.

Object Atomicity Files must not be modified by the system beyond the modifications required for the synchronization.

SHOULD Have Features in this category are features that built on the MUST have features and are thus not strictly required. In broad terms this includes two important aspects. First and foremost is the capability of having a peer that only retains an encrypted version of the data. Built on this the second aspect is the server client that only retains an encrypted data set of a user's Tinzenite directory. The server also adds the capability of handling multiple users' data on a distributed file system capable of handling the large data size that is to be expected for multiple concurrent users.

Third Party Client The capability of supporting encrypted clients should be implemented.

Protocol Extension To enable an encrypted third party client, the core protocol will have to be expanded.

Performance Working on an additional peer type should allow us to improve performance of the protocol.

COULD Have These features are features that will only be implemented if all previous features have been successfully integrated. They are not required for the thesis to be considered successful but would be nice to have to fully complete the proposed functionality. Primary aspects that would be added in this phase are additional clients with differing functionality: a mobile client for Android, a web interface for accessing encrypted server clients, and a passive storage client. These would require additional protocol extensions to support higher performance and better control over the synchronization, such as shadow files and delta updates.

Shadow Files Peers could be allowed to only fetch files that the user explicitly wishes to have synchronized on a peer basis.

Delta Updates Transfer times of files over limited bandwidths could be optimized by only transferring the differences between them.

Web Interface Support for accessing an encrypted peer via a website.

Mobile Client Implement a client that can run on an Android device.

Passive Peer Built on top of the functionality required for the third party client we could also implement the capability that Tinzenite can use passive storage as passive encrypted peers.

3.3 Dependencies

This section will introduce the software and programming language this thesis will be based on. We will include a brief overview of each technology and highlight some points that are important to our work. A more technical discussion on these technologies can be found in section 5.1.

3.3.1 Golang

Our implementation language of choice is Golang, usually referred to by the shorthand Go [Golc]. Since the main language at Ulm University is mostly Java for student work, this thesis will hopefully also offer some insight into the differences between the two.

Golang was first created at Google in 2007, but announced and open sourced in 2009. The reason for yet another language is given in the language's frequently asked questions page as: "Go is an attempt to combine the ease of programming of an interpreted, dynamically typed language with the efficiency and safety of a statically typed, compiled language. It also aims to be modern, with support for networked and multicore computing. Finally, it is intended to be fast: it should take at most a few seconds to build a large executable on a single computer." [Gola]. Since Tinzenite is a networked system and offers many possible applications for using concurrent operations it was deemed a good match for this thesis. More information on the origin of the language and its design goals can be found in this article [Pik12].

3.3.2 Tox

A core aspect of this thesis is implementing the system using the peer to peer encrypted communication channel provided by the Tox communication suite [Toxa]. Initially developed as a Skype replacement the underlying transport layer was also intended to be usable for alternative services. We will make use of this and let Tox handle most of the communication aspects.

Unlike many other communication applications, Tox does not require a user to register an account somewhere. Instead a user can create as many Tox identities as desired locally. Each identity consists of a public and private key where the public key is the identity of the user [Toxb]. Tox identities are dynamically mapped to the user's current internet address whenever the users are online via a distributed hash table. Once two users are online at the same time[2] and have added each other as friends they can establish a communication channel. The Tox messaging clients use the channel to facilitate text, audio, and video chats, with support for file transfers. All data is encrypted with perfect forward secrecy and sent directly from one client to the other[3].

Tinzenite will build on the peer to peer, distributed, and encrypted communication channel provided. In our case each directory on each peer will be its own Tox identity. Every

[2]Tox is currently not capable of offline messaging. However the feature is planned to be implemented some time in the future.

[3]Although TCP relay tunneling is sometimes used to punch through obstructions.

directory that is synchronized between multiple devices has its own network of friends which consists of the group of authorized peers. For the setup Tinzenite will require the user to allow the initial connection to any single other peer. The friend list is then synchronized by Tinzenite between all peers automatically.

3.3.3 Hadoop File System

As the encrypted server peer is intended to run as a service for multiple parallel different Tinzenite peers of multiple users it will require some extra work into how to store the large amount of data. Since storing large amounts of data requires significant management work we will provide an implementation of the encrypted peer that can write its encrypted data to the Hadoop Distributed File System [Bor07]. This file system is part of Apache Hadoop software library for scalable distributed software [Had].

Utilizing the Hadoop file system brings us a few sought after advantages. It will allow the encrypted peer to be implemented without having to give much thought to the actual storage and retrieval of data. We can thus build on the Hadoop file system's high fault tolerance which in turn allows the encrypted peer to run on inexpensive hardware with low risk of data loss.

3.4 Software Scope

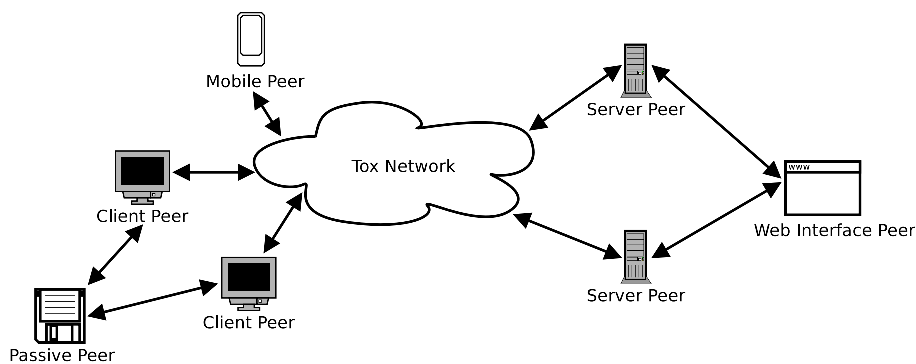

Figure 3.1: Example Tinzenite network with all proposed possible clients. Note that not all clients communicate via the underlying Tox communication network.

This section is dedicated to differentiating the possible client applications we will implement as reference implementations. Figure 3.1 shows an overview of how the various parts could mesh together to form a complete Tinzenite network. Note that the exact feature set is to be determined by the required development time of each feature and thus might lead to some features or even complete clients not being implemented for the thesis. Those features can be implemented at a later time if so desired and will thus be referenced in the future work (see section 6.2).

3.4.1 Tinzenite Core Library

The Tinzenite core library is the central reference implementation of the protocol. It builds directly on the Tox core library and wraps the complete communication of Tinzenite. Programs that implement the provided functionality will attach callbacks and call public methods to interface with it. The library will also handle watching a specified directory on disk. If the user modifies the contents of the directory that is watched by the library it will update its internal model and start trying to synchronize the changes to other peers.

To keep development of clients as easy as possible while at the same time keeping the protocol consistent between them we will separate the core logic for Tinzenite from any user oriented code. This will ensure a maximum of adaptability for clients, meaning that they will not be constrained by the cross platform capabilities of the library itself. The only limiting factor for porting the library to other platforms is the availability of the required Tox core library beneath it.

3.4.2 Client Peer

The basic client peer will be developed first and serve to validate the protocol. This software will be the target of the user's primary interaction with the Tinzenite system. Therefore we plan on including full coverage for required assistance in connecting peers and setting them up. The client peer will always be a trusted peer and thus store the user's data in clear text on the disk. Since the client peer will be the primary implemented peer, it will evolve directly with the core library. Advanced features that may

be implemented include but are not limited to support for shadow files and a low system footprint so that the client software can be run continuously without negatively impacting the operating system performance. From the user's point of view the client software will provide an interface to connect to and accept new and existing peers.

3.4.3 Server Peer

The second dedicated software to be developed is the server peer. This software will implement an encrypted peer for the Tinzenite network. As the task of encrypting data falls to the trusted peer before uploading said data, the server peer must only offer a simple key value storage system for writing the encrypted data somewhere where it can later be retrieved. Thus the encrypted peer allows clients to offer a storage interface and will use that instead of writing via a self specified method.

As a proof of concept that this is going to work we will develop two instances of such a storage interface. The first one will be a simple store and retrieval to the local disk where the encrypted peer is running. The other one will utilize the Hadoop file system to offer an example of a large scale distributed storage system as it might be used on an actual server for multiple users.

Unlike the client peer however we do not require user interfaces and file watchers. Since the encrypted peer must only be started and connected to an existing network, no further user interaction than those two operations are required. A file watcher like the client peer is not required because updates can not originate from the encrypted data set. These encrypted peers will also utilize a different subset of the communication protocol as they operate essentially as blind data dumps.

3.4.4 Web Interface Peer

Built on top of an encrypted server peer it should theoretically be possible to implement access to the encrypted data for the user via a web interface. This can be realized by allowing users to enter their password which in turn unlocks the decryption keys required to decrypt the stored data. By utilizing client side code within a web browser, these

operations can be done without involving the server peer beyond its use as a blind data store. This peer would require a Javascript implementation of the Tinzenite protocol. Golang can be compiled to Javascript [Gitc] which should allow at least some reuse of existing code.

3.4.5 Mobile Peer

Nowadays no software is complete without a complimenting mobile application. Thus Tinzenite should also offer a mobile client for the Android platform [And]. Apart from the specific touchscreen oriented interface this peer would prove that Tinzenite can run on mobile platforms. This would build on Golang's cross platform support even for Android [Golb]. Further work may be done here to ensure that Tinzenite runs in a mobile friendly way. This implies low power consumption and low bandwidth capabilities. Therefore support for shadow files and deltas is almost strictly required to meet the needs of a mobile peer. When first using the mobile peer for the first time it may even be beneficial to ask the user whether the mobile peer should pull all data or just immediately mark everything as shadow objects, thus requiring the user to mark files manually to be fetched locally but reducing bandwidth requirements enormously.

3.4.6 Passive Peer

While not in itself a program, support for blind data dumps as passive peers is another aspect for which Tinzenite could be expanded to include support for. This feature would require of Tinzenite to run the base encrypted peer logic on a given directory without the communication aspect via Tox. These passive peers would allow easy backups of directories on passive storage media that can be easily updated by connecting it to an active peer.

4 Architecture

The following section will define the data model each peer keeps of its data and specify the API for updating it between peers, both trusted and encrypted. Therefore we start this chapter with an explanation of all meta information that a trusted peer stores, as everything else is based on top of this information. Then we will define and explain the model each peer keeps of the stored data. Built on this model we will discuss the messages used to facilitate the update of the model between clients. Then we will discuss the protocol extensions required for the encrypted model. Finally we will take a look at the advanced features and how they can be built on top of the previous work.

We chose to use JSON for all communication between peers as further discussed in section 5.1.2. Therefore generally speaking all messages will be defined as JSON messages. Additionally most data is written to storage as JSON files. This has the advantage of allowing easy manual access to all files and message for development purposes since JSON messages can be sent as simple text messages via the normal Tox chat clients.

4.1 Meta Data

Since a trusted peer must store the last known state of the directory to detect changes for later synchronizations, we must store this information to storage somewhere. For Tinzenite we took a page from Git and decided to write this information within the directory to be synchronized. To reduce visual clutter and to hide it from users this directory is offered by a hidden directory directly below the root directory. We specified

it as *".tinzenite"*. In short it contains organizational files, temporary objects, and the deletion directory for storing files until they have been fully removed by all peers, plus data private to the peer. Figure 4.1 gives a broad overview of the contents.

- .tinzenite/
  - org/
    - peers/
    - auth.json
  - removed/
  - temp/
  - receiving/
  - sending/
  - local/
    - rmstore/
    - model.json
    - self.json
  - .tinignore

Figure 4.1: Overview of the special meta information directory that Tinzenite uses to store relevant information required for the managing of the directory. For brevity sub directories that contain instance data are not expanded.

Placing the meta data within the directory has a few benefits for the complexity of the system and enables it to utilize the full feature set of the implemented synchronization capabilities to share data between peers when required. Specifically only the *"org"* directory and the *"removed"* directory are synchronized just as any user data, along with the *".tinignore"* file which specifies to ignore all other objects in the meta directory. The choice to use Tinzenite itself to handle these directories and files was done as they are required for trusted peers to work correctly. The *"org"* directory is included because it contains both the authentication file and the peer files – both are required to be kept synchronized between all other trusted peers. The *"removed"* directory is included due to how removals must be handled as discussed in section 4.2.3.

4.1.1 Organizational Directory

The *"org"* directory consists of two objects: the file containing the authentication information and a sub folder which contains a file for each known peer. It is first of two directories that are synchronized along with user specified files between peers, as both the authentication file and the peer list are required for all peers to function correctly.

Authentication File

The authentication file stores information required for the complete Tinzenite network. This includes information on the user, information on the directory the network synchronizes, and the encryption keys required to encrypt and decrypt data for encrypted peers. Since the authentication file is not encrypted upon upload to encrypted peers as it is meant to be used to enable user account management, all personal or otherwise critical information is stored either hashed or fully encrypted. Listing 4.1 shows the contents of an example authentication file.

```
0 {
1     "User": "$2a$10$E8Wlr9Jn/EYJLZ7J0yZoR.Qscp.MKD2kG8dHF7OQWYNA1mCfp.Qqe",
2     "Dirname": "sync",
3     "DirID": "ffff1d4cbfded232",
4     "Secure": "Kbk4+sx17VKHma1Z67OU6R7TbHPWMr4SpZhWUQqheS/CNcKKHVYjTTSv0rbF4qDAa0
          vwikigsm7wHhy4iGjWB84i0ErO7rNwhqrPPxudeDM=",
5     "Nonce": [255,142,165,173,201,188,98,116,29,31,173,181,84,84,137,54,159,50,193,248,
          51,162,76,195]
6 }
```

Listing 4.1: An example of an authentication file.

The key *"User"* stores a bcrypt hash [PM99] of the user's name. This is important for example for the support of encrypted third party peers: they can attach accounts to the provided user name for controlling server side access. It provides a way to enable encrypted server service providers to attach their own accounts directly into a Tinzenite network so that access can be managed. The user given name of the directory is stored in *"Dirname"*. It is not hashed so that the user can easily read which directory the current authentication file belongs to. We also need a way to distinguish multiple synchronized directories from each other: this is the random unique hash stored in *"DirID"*. This

unique identification also allows us to differentiate multiple Tinzenite networks that might share a directory name. Again, this can be used by third party service providers to differentiate the amount of directories that a user can store with them. The encryption keys for encrypting and decrypting file data for encrypted peers is stored in *"Secure"*. These keys are encrypted with a password derived encryption scheme further discussed in section 5.4.2. *"Nonce"* stores the nonce value that is required additionally to the user provided password to successfully access the encryption keys.

Peer Files

Within the *"peers"* folder Tinzenite writes all data related to synchronization peers. It is in essence the contact list, filled with information required to access peers. This includes a peer's address, trust, and further user defined information.

```
0 {
1     "Name": "box",
2     "Address": "b6ad2388839d3068f9d6562c10d1151dd87818373c88cf9aad829144c63aac36",
3     "Protocol": 1,
4     "Trusted": false,
5     "Identification": "19baf5873da66797"
6 }
```

Listing 4.2: An example of a peer JSON object. Note that the encryption attribute is not to be trusted, it is only an optimization.

Listing 4.2 shows the structure of an example peer file. One of these must exist for all known peers. The *"Name"* is the user defined name for each peer, to be used by the user to make differentiating between peers easier. Internally however the peer is referenced by the random assigned hash stored in *"Identification"*. The Tox address is stored in *"Address"*. Whenever a new peer is added and its peer file distributed to all known peers, they can use this information to automatically accept connections to the new peer. The *"Trusted"* attribute is an optimization: peers for which this value is false don't need to receive and decline a challenge. It is important to note that the other way around is not true: if the value is false the other peer must still respond successfully to a challenge. Finally the *"Protocol"* value defines via which protocol the peer can be reached. This is currently unused as we only use Tox.

4.1.2 Removed Directory

- ⊸ removed/
  - ⊸ db198086d1708794/
    - ⊸ check/
      - ⊸ 927325d7a930dac9
      - ⊸ ec021135799691ae
    - ⊸ done/
      - ⊸ 927325d7a930dac9

Figure 4.2: Example of the removed directory with an active removal pending. Note the missing file in *"done"* which would signify the completion of the removal.

The *"removed"* directory stores objects that are pending removal as described in section 4.2.3. An example of the directory with a pending removal can be seen in figure 4.2. Notably this directory is synchronized among Tinzenite peers just as any data the user synchronizes. This is due to the fact that removals must be synchronized by all peers.

For each object that is removed a directory with the object's identification is created. Within this directory two sub directories are created in which empty files named after peer identifications are placed. In the *"check"* directory peers write a file named after each peer who must confirm the removal. The *"done"* directory contains a file named after every peer that has applied the removal and is now also waiting for it to complete.

4.1.3 Unsynchronized Directories

All other directories within the *".tinzenite"* directory are not synchronized between peers as they are only used to store locally relevant data. The file which contains the rules for this is the *".tinignore"* file (for more information on ignore rules see section 5.3.1).

The *"temp"*, *"received"*, and *"sending"* directories are used for files in transit either in preparation before being sent encrypted or upon receiving before being applied to the local directory. Files are transmitted by chunks by Tox. Therefore, to keep the RAM storage requirements down, these blocks are immediately written to storage. This also

avoids partially overwritting user accessible files if the transfer fails for some reason, thus ensuring that Tinzenite only overwrites the user's files when it is ready to do so with a complete file.

The *"local"* directory is used for three purposes. First a copy of the peer's own peer information is written into a file alongside with a binary dump that the underlying Tox channel requires to run which contains the persistent state information. Second the actual model is written to storage here, although not as a fully expanded object tree as specified in section 4.2. Instead Tinzenite writes the internal representation of the model to storage as JSON. This differentiates in a few key points, but primarily serves to store both the absolute path and to keep the directory tree in a flat representation that is more efficient to actually work with. Finally the *"rmstore"* directory keeps a record of all previously completed removals in the case that a peer tries to reintroduce a completed removal.

4.2 Object Model

This section will describe our solution to how Tinzenite keeps track of objects within a directory. This data will be henceforth referenced to as the data model or just model. Based on this we will highlight how Tinzenite applies object operations on the model in a further section.

The model is required to enable detection of newly created and removed files, since Tinzenite does not actively watch a directory. Having a stored representation of a directory significantly eases the difficulty of detecting file creations and removals, even if the peer software is not running. Any entry in the model is generally referred to as an object if the distinction between a directory or file is not required.

An important feature that the model should have is that it should represent an arbitrarily complex object structure in the most simple way possible. Therefore there are only two assumptions we will make for the structure of any directory: namely that it contains files sorted in nested directories. Out of this tree view we can immediately synthesize our two main components that we will require: a file model (a leaf) and a directory model (a

node). Since a peer is intended to have a directory as the root node from which to run, the core element will always be a directory. An example of the proposed model structure can be seen in 4.3.

- ⊸ Root Directory/
  - ⊸ .tinzenite/
  - ⊸ Sub Directory/
    - ⊸ File
    - ⊸ File
  - ⊸ File
  - ⊸ File

Figure 4.3: An example of how a data model of a directory is structured. The .tinzenite directory is discussed in section 4.1.

Since each file is considered a binary blob and must not be modified by Tinzenite in any way to preserve data integrity, any additional information that Tinzenite is required to store for an object must be kept within the model itself. Out of this we can see which values need to be stored within the model for each object specifically.

Each object in the model will be specified for identification purposes by a unique randomly generated hash. This hash allows us to decouple the name of the object from its model representation, effectively serving as the same function as node identification numbers when stored on a hard drive. Furthermore each model object will contain a path variable that specifies the relative path of the object in the directory tree. This has the purpose of allowing the placement of all files in the correct locations on storage for a given root path.

```
0 "Version": {
1     "927325d7a930dac9": 1,
2     "19baf5873da66797": 4,
3     "ec021135799691ae": 3
4 }
```

Listing 4.3: An example of a version vector clock. Each hash is the identification of a trusted peer, the associated number which version of an object they last modified.

Apart from the above attributes Tinzenite must also track versions of files to allow detection of when objects have been updated. We will use a vector clock [Mat89] to implement this, where entries represent peers and the associated number is the last version where that peer actively contributed to the object's history. The vector clock can also be used to detect collisions. Note that the vector clock must only store the versions for active, trusted peers as these are the only peers where versions can differ upon user interaction. We avoid using a simple dirty flag for reasons of complexity: determining which peer's update to take in which order is not trivially doable with a simple boolean flag. Utilizing vector clocks gives us greater flexibility, both for the implementation and for any visualizations in the case of tracing changes. An example of the vector clock as we will use it can be seen in listing 4.3.

It is important to note that the model will not be used to store peer reliant information. This includes for example where the directory is placed on the peer's file system, which may differ between peers. Such information must be stored separately by the peer and be applied when working with the data model, for example when determining what the full path on the file system will be for a file that is to be written. Some properties are also not suited to be transferred between peers. This includes file system or operating system dependent properties such as usage rights, ownership, or flags. For Tinzenite we will generally ignore these as the primary focus is just raw data synchronization without semantic information.

4.2.1 Directory Model

```
0 {
1     "Directory":true,
2     "Identification":"db198086d1708794",
3     "Name":"test",
4     "Path":"test/test",
5     "Shadow":false,
6     "Version":{},
7     "Objects":[]
8 }
```

Listing 4.4: An example of a directory JSON object. Note that for brevity no files or sub directories are shown in the *"objects"* array. The version object is also left empty here.

Listing 4.4 shows an example of the proposed JSON structure for representing a directory. A directory is somewhat special as it does not require the synchronization of an attached binary file. This is the case for Tinzenite because directories are viewed independent from their content.

The *"identification"* attribute is a random generated hash that uniquely identifies the directory. The *"path"* attribute stores the concatenated relative full path from the peers root directory to where the directory lies. The clear text *"name"* is also stored here as an attribute. The *"shadow"* flag is used to signal whether the contents of the directory are to be fetched or not. To differentiate between updates we require a *"version"* attribute which represents a vector clock of peers and their last known version.

Finally an *"objects"* array is where the corresponding sub directories or files are recursively placed. To model a directory as shown for example in figure 4.3 Tinzenite begins the model with a directory model for the root directory. Within the objects array one can the then find the two files and the two further directories. Each directory object in turn also stores sub objects in its object array, thus recursively modeling an entire directory.

4.2.2 File Model

```
0  {
1      "Directory":false,
2      "Identification":"b83cf06d4e056e1a",
3      "Name":"else.txt",
4      "Path":"test/else.txt",
5      "Shadow":false,
6      "Version":
7      {
8          "927325d7a930dac9":1
9      },
10     "Content":"e4abda92f30700d751ac82f7454787d5"
11 }
```

Listing 4.5: An example of a file JSON object.

Listing 4.5 shows an example of the proposed JSON structure for representing a file object. The *"identification"* attribute is a random generated hash that uniquely identifies the file. The *"path"* attribute stores the concatenated relative full path from the peers root directory. The clear text *"name"* is also stored here as an attribute. To differentiate

between updates we require a *"version"* attribute which represents a vector clock of peers and their last known version of this file. Important for detecting file changes is the *"content"* attribute which stores a hash of the file's binary blob. Finally the *"shadow"* flag is used to notify a peer whether the file is locally accessible or must first be fetched from other peers.

4.2.3 Object Operations

Based on the defined model we will now discuss which file or directory operations are applied in what way to the model. Tinzenite relies only on the most basic file operations for manipulating both the model and the actual file directory. Therefore we require only the following four operations for the basic case to work:

Create Created files are detected by simply noticing files that do not exist in the model yet and are not listed as removed. Files that have been created will be added to the model at the correct location and their attributes calculated, if not given. Tinzenite then checks whether it needs to fetch the file if it wasn't created in its own directory instance.

Modify Modification is either when the model does not match the file anymore, in which case a new file must be fetched, or the content of the local file has changed. This is detected via the content hash[1]. The model is updated to match the file again.

Remove A removal is detected when a file does not exist anymore and was previously tracked by the model. File removing is one of the most complex cases in Tinzenite due to the insert delete ambiguity. We will solve this by storing the models of deleted files until the delete update has been propagated to all currently known peers. Only then the model is also discarded. For this to work Tinzenite must always ensure that files that exist but are listed as deleted are not added back to the model as a new file by continuously checking the deletion list.

[1] Since hashing a file is an expensive operation we have also implemented that the hash is only checked if the modtime of the file has changed since its last update. This greatly speeds up checking for modified files.

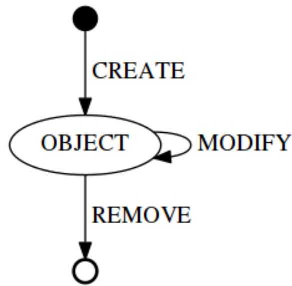

Figure 4.4: The life cycle of every object, file and directory, in Tinzenite with the allowed operations.

These three operations are not the only operations that a user can do on directories or files. However they are all the operations that Tinzenite requires so that it can cover all possible file state transitions. The choice of these three basic operations for Tinzenite was heavily based on the paper discussed in section 2.2.2.

Therefore the life cycle of an object is very simple, as can be seen in figure 4.4. Objects can only come into existence via the create operation. Objects can either be changed by changing their content which is a modify operation, or by changing their location which is a move operation. Finally objects can also be removed. If an object has been removed it will stay removed unless the user creates a new clone of it.

In the following sub sections we show the actual definition of the specification and expand on how Tinzenite peers react to them in more detail. This specification makes up the core functionality of how the data synchronization happens within Tinzenite. Note that these operations are only for trusted peers: since encrypted peers can not work on the stored data they have no need for the file operations.

Create

The creation of an object can be trivially detected in Tinzenite. Basically there are two cases where an object must be created: the first is local manual creation which implies that the peer is the origin of the file; the second is creation via receiving a creation message from another peer.

In the first case, if the object does not have a representation within the model and is not listed as removed, it triggers the creation case. Tinzenite creates the correct object representation and inserts it into the correct place in the model. This includes creating the unique identification hash and, if the object is a file, creating a hash of the file contents. For a remote creation the peer will queue a fetch operation for the required file. Only when the file has been received completely it is placed at the correct location and the model will be updated.

Modify

The modification of an object is not as trivially detected as an object creation. Special care must be taken in the case of conflicting changes which can obviously happen since the directory is shared among multiple peers (see section 4.4). Here we will only consider what happens once a modification has been detected. Again we have two possibilities for triggering this case: a local file has been modified or the peer has received an update message for a remote modification.

If the modification is detected locally we update the model to match the new directory state for that object and then initiate an update message. If, on the other hand, we receive a remote modification, we queue the fetching of the updated file. Upon receiving the changes we apply them and finally propagate the update to the other connected trusted peers.

Remove

Finally the deletion of an object is the hardest case within Tinzenite because of the insert delete ambiguity as discussed in section 2.2.4. It is nontrivial to ensure that an object deletion has been received by all required peers so that it truly and finally can be removed from the complete Tinzenite network. A simple solution would be a list of all deleted objects ever, but this list would promise to grow quickly on often used synchronized directories. Therefore we require a sort of garbage collection so that we can trim the list from deleted files that have been applied to all known peers.

We will look at Tinzenite's deletion of an object that has been removed locally first, then discuss how the update propagates and what the other peers are required to do to ensure a safe and complete deletion. Upon detection of a local deletion the trusted peer creates a directory within the *"removed"* directory with the name of the removed object's identification hash. Within this directory the peer copies the list of currently known trusted peers[2] to the *"check"* sub directory. We utilize simple empty files named after the peer identification hashes as entries since that is all the information we require for a deletion. We will refer to these files as the peer entry files for the purpose of describing the rest of the operation. To complete the removal operations required to initiate the removal through the Tinzenite network, the peer writes another peer entry file to the *"done"* sub directory. This sub directory contains a peer entry file for all the trusted peers that have acknowledged receiving the deletion update. For an example of a removal directory for a file, see figure 4.2.

Since the removal directory is also synchronized to all trusted peers they can continue applying the removal. Each peer receives first the creation of the removal directory and the deletion message for the object. Once the removal has been locally applied each peer writes a peer entry file to the *"done"* directory – so long as it finds a corresponding file in the *"check"* directory. If the peer knows of peers that do not have a peer entry file in the *"check"* directory it must append them. This guarantees that all peers that have known of the existence of an object will receive its removal. An example of where this appending is vitally important is when a new peer has recently been added to the Tinzenite network but its creation has not yet reached the deleting peer. In this case at some point the removal and the creation of the new peer will collide and the peer where the collision happens will append the newly created peer to the *"check"* directory. This guarantees that even though the peer where the removal originated from did not know of all peers, it will still wait for the new peer to also apply the removal. This is guaranteed to happen as the new peer entry file for the appended peer will reach the peer where the removal originated from before the removal is completed in any case.

[2]Note that encrypted peers must not be considered because they basically mirror only active peers, meaning they are guaranteed to correctly apply deletions once the active peers have reconciled it.

Once the last peer enters itself into the *"done"* directory it propagates the update one final time and can then deletes the removal directory. It is important to note that this deletion differs from other tracked removals: unlike when any other object is removed, deletions within the *"removed"* directory must not trigger another round of tracked removals. This would cause an endless removal of removals and basically bring the entire Tinzenite network to a halt as each peer is swamped with keeping up. Instead the objects and associated model entries are simply silently purged. Yet since it is very likely that not all peers delete the removal directory at the same time we must guard it against being unnecessarily reintroduced. Therefore each peer must locally remember each deletion for a specified time span until it can be sure that all peers have applied the completed deletion. During this time span each peer ignores the reintroduction of the deleted objects[3].

Now a final note on what happens should a peer receive a deletion update for an object that it doesn't know. In this case the message can not simply be discarded silently as there is no other representation of the update in the object model. This could lead to orphaned updates if the peer acts as a bridge between two peers that were previously directly connected. Therefore unknown deletion updates should be propagated if possible. However, since orphaned updates can only happen as long as the peer list has not been fully updated between all peers, this is a sufficiently unlikely case that we can live with. By adding a time stamp we can even detect orphaned deletion updates and warn the user to help solve the issue. Alternatively we can also warn if we have too many possibly orphaned updates, although this would signify a larger issue. Since there is no way to reliably ensure that all peers are in the same state simply for the reason that a peer may be offline at any point, there is little more we can do to ensure clean removals. It is more important to work towards a common model state than to ensure that previous states were legitimate for all peers.

[3]As an improvement to this behavior we also utilize notification messages to more rapidly terminate removals. See section 4.3.5 for more details on this.

4.3 Communication Specification

This section describes the specifications for the messages that will be used to synchronize two models on separate peers. We start with defining how connections are managed, then build on this to explain how new peers are added to an existing network. Then we will specify the messages used for model synchronizations between trusted and between trusted and encrypted peers. Finally we will briefly expand on the ordering of messages and the expected emerging behavior.

Every peer views all connected peers as separate connections. A swarm behavior only comes to pass because of the independent actions of every peer, not through a combined communication between multiple peers. This primarily makes it relatively easy to implement a Tinzenite peer, as the communication state is never between multiple peers. Therefore a single peer has a base state of no other connected peers.

4.3.1 Connection Management

In this section we will discuss how Tinzenite connects to other peers. As we distinguish between trusted and encrypted peers we must determine how the data we will send is to be modified accordingly. Note that encrypted peers by default will not initiate connections, only trusted peers will do that as they are the only peers capable of working on the user's data. Encrypted peers in general have a very passive role in the Tinzenite network.

Since each trusted peer has access to a list of all connected peers (see section 4.1.1) this list is used for the initial presumption which connected peer is trusted and which is encrypted. Therefore peers marked as encrypted are not even issued a challenge to authenticate themselves. To vet trusted peers however the challenge must be issued and met. Consequently the trusted peer queries each trusted connected peer in turn. Since we should not trust a peer just because it has been marked as trusted, this query must be cryptographically secure. An attacker that wants to connect to a trusted peer illegitimately thus can not establish a clear connection without the required keys, which the attacker

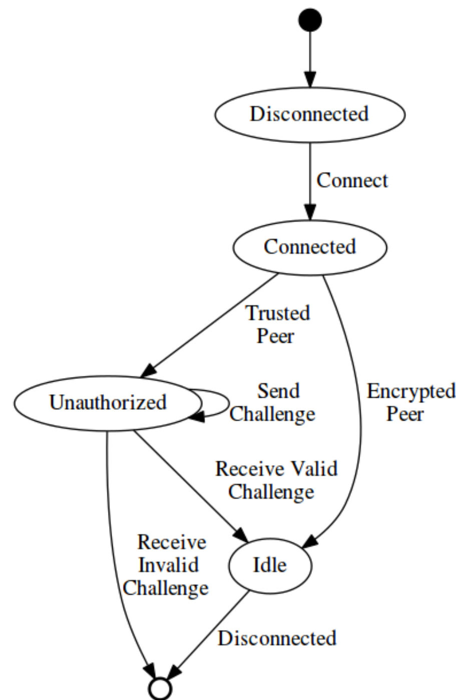

Figure 4.5: This diagram shows how peers handle the establishment of a connection to a known peer.

should not be in possession of[4]. We define this as the authentication challenge. Only if the challenge is successfully answered the other peer is considered trusted and the following communication will not be encrypted. If the challenge is answered incorrectly or not at all we can not be assured that the other peer is either trusted or encrypted. Therefore we simply ignore it for all further operations[5]. This concludes opening a communication channel with a connected peer.

Figure 4.5 shows an informal state diagram of how a peer reaches the idle state where the connection is ready and set up, both for trusted and encrypted peers. The switch from the disconnected state to the connected state is signaled by the underlying Tox connection, meaning it happens when the other peer is online and visible via the Tox channel. Once connected and if the peer itself is a trusted peer, a challenge is sent

[4] Tinzenite is designed to avoid revealing keys to outsiders. However attacks such as key loggers are beyond its control.

[5] Ideally we would warn the user of this as it may result in peers being orphaned within the network. Since this requires either the user having tried to connect to an insecure peer or a peer having become compromised, the chance that this behavior is undesired is comparatively small.

which contains an encrypted nonce. For a more comprehensive look at the challenge response mechanism see section 5.4.3.

Note that a few points must be considered that can not be shown in the diagram. Since Tinzenite is designed as a peer to peer network, no client-server structure exists. This poses the challenge of who begins the construction of a connection, especially as we can not distinguish between peers that have been online for a while already and peers that just become activated when the base peer connects to the network. This is due to peer to peer architecture of Tox: not all peers will respond to queries within the same time frame. From a network perspective peers that are further away will seem to be online slower than peers that are nearer. We solve this issue by having the peers use a random back off time whenever a request expects an answer[6].

Since we can not trust the other side of the Tox channel to be a Tinzenite peer, the peer initiating a connection will not advance beyond the step where it expects an answer. This means that for example an attacker would either receive only a challenge (to which he can't respond without knowing the keys) or the request for the meta information block to which he'd have to answer correctly.

A short note on closing a connection: once established this happens when the underlying Tox channel is terminated. Furthermore if something interrupts the establishment of a connection, both peers will simply go back to the starting point. Establishing a connection is done every time a peer reconnects.

4.3.2 New Connection Establishment

The previous section described how Tinzenite connects to known peers. This section therefore discusses how Tinzenite connects to new peers, meaning peers that the user adds to the network. Notably this is a manual process where the user is used as a secondary channel to prevent man-in-the-middle attacks. Since a typical Tinzenite network might have many peers, it is not desirable to have to authenticate a new peer manually with every other peer. We therefore propose that the user only needs to

[6] For example the challenge message for establishing a trusted connection.

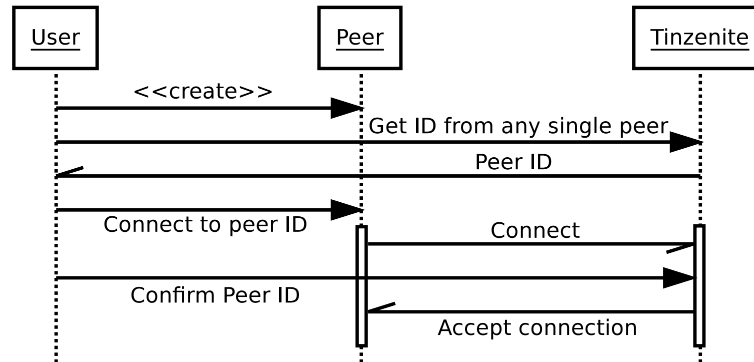

Figure 4.6: This diagram shows the interaction required to connect a new peer to the existing Tinzenite network.

authenticate the new peer with one existing trusted peer. From there the authentication is synchronized to the other existing peers much like any other file or property update. This bootstrapping process allows a user-friendly setup of peers.

Figure 4.6 shows how a user interacts with a Tinzenite peer and the Tinzenite network (consisting of 1 to n other existing peers) to connect a new peer to the network for the first time. As prose: the users simply start the new peer client for the first time, whether encrypted or trusted. Then they can point it at the directory location, change settings, and check whether a trusted peer is available. To establish an initial connection a Tox ID of an available Tinzenite peer is required. The users can then command the new peer to connect to this available peer by entering the Tox ID. To ensure that no man-in-the-middle attack can happen the users will now have to confirm to the connection the available peer. This means allowing the connection there and ideally ensuring that the seen Tox ID is identical with the new peer's Tox ID. Tinzenite then reports to the user whether the newly connected peer requested can be registered as a trusted peer or not. Once the connection has been confirmed for the given trust level on the other peer the channel is opened and ready.

What happens next differs somewhat depending on whether the new peer is an encrypted or a trusted peer. Since encrypted peers are passive peers the bootstrapping process is finished once the peer has established a connection to a trusted peer. However the encrypted peer at this state still lacks any encrypted files or more importantly the list of other peers and the authentication file. It falls upon the connected trusted peer to

initiate the first data upload, after which the new encrypted peer will have all the required information.

If the new peer is a trusted peer it must autonomously complete the bootstrapping by fetching all the files from the existing peer. This works even without an authentication file because the existing peer has received user confirmation of the trust status of the bootstrapping peer. This allows it to fetch all files unencrypted. Once all files have been fetched the new trusted peer is ready to run. This includes immediately connecting to other existing peers since the peer list is now also available to it.

Even if the new peer goes offline immediately after completing the bootstrap process, the new peer information will be synchronized throughout the already connected peers once they synchronize with the peer which accepted the new peer. This is possible because the connection request by a new peer includes its own peer information, and when the connection is accepted this peer information is immediately added to the *"peers"* directory. Thus it will be synchronized to all other peers who will automatically accept any connections of peers listed in the peer list.

Removing peers will work much in the same way from a system perspective. A user can note a peer to be removed, resulting in it being removed immediately from the peer where the action was initiated. The removal will then transit through online peers and result in a complete removal once all peers have been online. An interesting aspect to how to handle removal from the perspective of the removed peer is what to do with the remaining data. For a trusted peer we might offer a choice whether to remove the data or simply disconnect it from the Tinzenite network. On the other hand encrypted peers can simply remove the data immediately as they can't access it anyway. Notably removing a peer requires no action on the removed peer's side. This ensures that a peer can be excluded even if direct access to the peer was lost.

4.3.3 Message Order

We must define which messages are required to be sent when to keep two models between peers synchronized. Once the connection is established we must differentiate

between trusted peers and encrypted peers. For trusted peers a connection is only considered established after the Authentication Messages have been exchanged.

For trusted peers we propose the following message timing to enable rapid propagation of updates throughout the network. Upon connection both peers will first exchange models, providing essentially a snapshot of differences since the last synchronization. This requires a message to request the model from the other peer. Since we will also need to request files we use a generalized Request Message that can trigger a file transfer. Upon reception of the model, both peers will merge the models and request any changed files. At the end of a model request both peers should share a common model state and associated directory state.

If local changes happen during or after the model has been exchanged the Update Messages are used to notify the other model of the update without having to retransmit the entire model again. However to ensure that all updates do propagate at some point, ideally rather faster than longer, we propose that each peer resends the model every few units. The exact spacing of when the complete model is resent is again up to the peer itself and can be adjusted per instance. This allows mobile peers to work more energy efficient by synchronizing less often in comparison to desktop peers where power and bandwidth usage don't play such prominent roles.

Since encrypted peers work passively it is again up to the trusted peers to ensure that they are queried regularly. To avoid creating merge conflicts on encrypted peers which these could not independently resolve since all data is encrypted, encrypted peers can only be accessed by a single trusted peer at the same time. This will be negotiated via Lock Messages. Once successfully locked request messages can be used to fetch data from the encrypted peer. To place data on the encrypted peer we thus require a Push Message.

4.3.4 Trusted Synchronization

Now that we have discussed how a peer is structured and how its connections are handled, we can turn to how a trusted peer receives and sends messages via the Tox

channel. Once the setup has been completed, as seen in section 4.3.1, the models must be updated if they do not match. This happens normally when files and directories have changed through user interaction.

Authentication Message

The authentication message is used to verify trusted peers against each other to ensure that clear text data is only sent between themselves. The same method is used for both challenge and response. For a more in depth look into how the mechanism works see section 5.4.3.

```
0 {
1     "Type":"challenge",
2     "Encrypted":"5HFXCQBMwe6ohx9fKeLsNxaBsOK7+9pxWFiXX4aE4cRtdt8oUOHQBJ6XrowiwwgLunM="
3 }
```

Listing 4.6: An example of a valid authentication message containing a challenge.

Listing 4.6 shows an example of a challenging authentication message. The *"Encrypted"* value contains the challenge value or the response value encrypted with the data encryption keys.

Request Message

In general each peer does not rely on other peers to update. Therefore we propose the creation of request operation which serves to request resources required for synchronization with another peer. This request can be used both for the complete directory model as for each specific file. It is important to note here that directories should never be requested as they do not require attached binary data to be either created, modified, or removed. Thus a request operation is only for operations that require data to be transferred. Upon receiving the message the other peer starts a Tox file transfer which the initiating peer knows to accept as it just requested it.

Listing 4.7 shows how a message to initiate a file transfer is structured. *"ObjType"* can have the values object, peer, model, or auth. The peer and auth values are used by

```
0 {
1     "Type":"request",
2     "ObjType":"object",
3     "Identification":"b83cf06d4e056e1a"
4 }
```

Listing 4.7: Example of a JSON request message.

the encrypted peer to differentiate these special meta data from normal encrypted files. *"Identification"* is the unique hash that each object is referenced by.

By switching the *"ObjType"* attribute from object to model the other peer, whether encrypted or trusted, can be commanded to send its current model. After comparing the received model to its own model, the peer knows for which files it must request data. Thus actively requesting a full model update periodically will still allow updates to propagate independent of individual update messages at some point in time.

This message is also explicitly used to get an encrypted peer to initiate a file transfer of the requested file or the encrypted model. The encrypted peer will respond with a file transfer of the requested object if the trusted peer has successfully locked it. If not, the trusted peer will receive lock denial messages until it has successfully locked the connection.

Update Message

Continuously requesting and receiving the model is not ideal. For large directories just the transfer of the model may require significant bandwidth. Therefore we have included the option of sending current updates to all currently connected trusted peers so that they can apply the update without first having to request the complete model.

Listing 4.8 is a message for broadcasting a model update to other connected trusted peers. *"Operation"* can be any value of either create, remove, or modify. The key *"Object"* contains the object representation for the updated object as seen in section 4.2.

The update message serves to keep both models in sync even after the initial synchronization at connection establishment. Upon receiving an update message the peer should immediately respond with a request message to retrieve the updated file. If a

```
0  {
1      "Type":"update",
2      "Operation":"modify",
3      "Object":
4      {
5          "Directory":false,
6          "Identification":"b83cf06d4e056e1a",
7          "Name":"else.txt",
8          "Path":"else.txt",
9          "Shadow":false,
10         "Version":
11         {
12             "927325d7a930dac9":3
13         },
14         "Content":"cecef410a6ef0d3c3e101ab726af2776"
15     }
16 }
```

Listing 4.8: The message broadcast by a peer to connected peers to notify that an update has happened.

peer receives a message for an operation that it has already applied, the message is silently discarded. This does not disturb the propagation of updates since the peer will have sent the update to the other connected peers if it already has the update. In this way we ensure that every message propagates as far as possible without consuming unnecessary bandwidth[7]. A positive side effect of the inclusion of the update message is that updates propagate through active trusted peers within the Tinzenite network faster. This in turn lowers the amount of time a user must wait for changes to arrive at another peer.

4.3.5 Encrypted Synchronization

Communication with the encrypted peer is a special case as all operations on the encrypted files must happen on the trusted peer. Since the encrypted peers can not resolve conflicts we must ensure that no conflicts ever happen specifically on the encrypted data. We do this by ensuring that only one trusted peer may synchronize with an encrypted peer at once by utilizing a locking mechanism.

[7]Peers are of course free to implement message sending in any way since the update messages are not critical for Tinzenite to work. Peers can simply only send update messages to a limited number of peers, down to none if bandwidth is scarce for example in the case of a mobile device.

Lock Message

```
0 {
1     "Type":"lock",
2     "Action":"request"
3 }
```

Listing 4.9: Message for negotiating a lock for a peer.

Listing 4.9 shows the format of the message used to negotiate a lock for an encrypted peer. The key *"Action"* can have a value of request, release, or accept. The trusted peer first sends such a request to which the encrypted peer can answer. If it answers with release or fails to answer the trusted peer considers the encrypted peer already locked and will retry at a later time. If however the encrypted peer responds with accept, the lock is active.

Once locked we must ensure that the encrypted peer will not become stuck in a locked state: we do this by having it implement a time out. This time out resets after every received interaction from the trusted peer so that we do not accidentally time out on long file transfers. If the trusted peer tries an operation while locked out, the encrypted peer will respond with a denial message every time. Once the trusted peer is done with its operations it should remove the lock by sending a release message.

Push Message

Since the encrypted peer is a passive peer, it requires a message to initiate a file transfer when files need to be uploaded to it. We solve this by using a push message.

```
0 {
1     "Type":"push",
2     "Identification":"b83cf06d4e056e1a",
3     "ObjType":"object"
4 }
```

Listing 4.10: Message that is used to initiate a file upload to an encrypted peer.

Listing 4.10 shows the format of such a push message. Both *"Identification"* and *"ObjType"* contain the same possible values as in similar messages.

When an encrypted peer is locked to a trusted peer and receives a push message it will respond with a corresponding request message. This request message will trigger the trusted peer to encrypt the file and send it to the encrypted peer. The encrypted peer will accept the file transfer because it now expects it.

For the peers and authentication file the different value of *"ObjType"* will allow the encrypted peer to place these unencrypted files in a different location from the normal data. This is required so that it can access them and work with the contained data as required. Since trusted peers store the model internally and the model is privacy sensitive as it contains clear text paths and names, the model is also stored as a special file on the encrypted peer.

Notify Message

The notify message is used to notify special states to peers. For trusted peers it can be used to signal that a removal has been fully applied to avoid reintroduction of removals. Encrypted peers use it to signal trusted peers that a request can not be answered if a file for a given unique hash identification can not be retrieved and transfered.

```
0 {
1     "Type":"notify",
2     "Notify":"missing",
3     "Identification":"b83cf06d4e056e1a"
4 }
```

Listing 4.11: Example notification message.

Listing 4.11 shows an example message for a missing object from an encrypted peer. The value *"Notify"* can be either missing or removed. The *"Identification"* is the unique hash for the object which the notification message references to.

4.4 Update Detection and Reconciliation

This section describes the theoretical side of the update detection: how we detect updates on file systems in a way that does not consume too many resources from the

hosting computer. Then we will discuss the interesting case of update reconciliation and how Tinzenite reacts to conflicts.

4.4.1 Update Detection

When a Tinzenite directory is created the initial model of the directory is created and stored as described in section 4.2. To detect changes within the directory Tinzenite thus regularly compares the current state of all objects within the directory against the previous model. If creations, modifications, or removals are detected they are applied to the model and any further required operations will be completed.

Creations are simply detected when objects exist in the storage directory that are not tracked in the model. The reverse is used to detect removals: tracked objects that do not exist in storage. Modifications must be checked for every tracked existing object.

4.4.2 Update Reconciliation

The core algorithm for how update reconciliation is done is another important element of Tinzenite. The following section introduces it and discusses the ramifications for the system based on the initial model synchronization while leaving aside the object update messages for now. These will be discussed once the basic mode is understood. Note that only trusted peers actively resolve updates, although reconciliation can happen for models received from both trusted and encrypted peers.

The reconciliation begins by receiving the model of the other peer. The model is temporarily stored so that Tinzenite can work with it until the reconciliation is finished. To guarantee that the directory remains consistent with the model right from the start we propose a simple rule: the model of the local directory is updated atomically with the successful reception of the binary object updates.

For each object in the received and temporarily stored model Tinzenite performs a series of checks. For the sake of decreasing algorithm complexity we start with the easy case of file creation.

Object Creation

Object creations are trivial to detect initially. Any object that the remote model has knowledge of that the local model has no equivalent entry for is considered as newly created. The naive check is simply whether for a given remote object entry an entry exists at the same path for the same identification. To avoid reintroducing objects that have been locally removed but where the removal has not yet reached the remote peer, we must also check this list of potentially created objects against all known local removals. If the creation is for an object that has been locally removed we ignore it as the remote peer is not up to date on this object. On the other hand if the creation is valid the local peer can request the binary data if the creation is for a file. Finally when the file has been received the creation is applied to the local model and the file written to storage at the corresponding location.

Object Removal

A removal is detected much like a creation, although the roles of the local model and the remote model are switched. Objects are considered candidates for being removed if a remote entry does not exist for a local entry. However we again have to counter the insert delete ambiguity. Thus removals are only considered valid if a corresponding removal directory has been created for the removal as described in section 4.2.3. If such an entry does not exist for a removal it is not a removal but a local creation that the other peer has not yet applied. Thus if the removal is valid the removal can be applied and the removal directory updated.

Object Modification

To detect object modifications we must check each entry that is neither created nor removed. Since the local peer does not have access to the remote file to detect if the file has been modified we must use some other mechanism to detect a modification. This is where the vector clock as described in section 4.2 associated to each entry comes

into play. For each candidate we compare the local vector clock state against the remote vector clock state. Tinzenite checks for every entry in the remote vector clock whether the local vector clock entry's value is equal or lower. If the entry exists and the value is equal or lower the local vector clock is current and the remote file has not been modified since the last check. If an entry does not exist or a version is higher the remote entry has been modified and we must fetch the file. What remains to be done then is to check for a merge conflict. We do this by simply reversing the comparison: if the remote version knows of all local version information it can be applied and the local file overwritten. If the states diverge both objects have been modified. In this case we trigger a merge conflict.

	Local object not modified	**Local object modified**
Remote object not modified	Nothing	Nothing (It is up to the remote peer to actively fetch this update.)
Remote object modified	Fetch remote object and apply update	Resolve conflict

Table 4.1: This table shows the four possible cases that can result from comparing the versions between two peers.

As stated Tinzenite uses only the model objects to detect changes, specifically only the version attribute. As shown in listing 4.3 the version attribute contains a list of peers and the last known version of the object they have. From this list we are only interested in the versions of the remote peer and the local peer. By comparing these two attributes between the versions that both peers have we can reliably detect how to proceed without losing data. Table 4.1 shows the four possible cases. Since there is nothing to do independent of our local state if the remote peer has no modifications[8], Tinzenite will check this first.

[8]This is the case because the other peer is responsible for requesting our local changes: the local peer has no obligation to actively push changes.

Conflict Reconciliation

As stated before Tinzenite considers files to be atomic entities and thus will not modify them itself. Therefore no automatic conflict resolution will be implemented in the base Tinzenite system. Thus Tinzenite must support the user in resolving conflicts. We do this by clearly labeling conflicts by appending a special word to the object names of the conflicting files. It is up to the user to actually resolve the conflict by editing, creating, or removing objects. So now we must store both versions of the conflict in a way that it can be easily resolved on any peer.

The solution we propose is in itself simple enough from a technical view. Basically, we remove the old version of the object and replace it with the two conflicting versions. As these are newly created files they can propagate through the network as any other object would, allowing the conflict resolution to happen even on peers that did not see the conflict initially. It is then up to the user to either keep both versions (likely renaming them) or merge the changes into one object and remove the other. Both cases will be propagated through the network as normal file operations.

4.5 Security Considerations

As stated in the motivation for this thesis in section 1.1 security and privacy are a primary concern for this thesis. We believe it important to give a brief overview of conceptual thoughts on the security of the proposed work. Please note that a full security review is not the goal of this work nor within its scope. Nonetheless we hope that the implementation will be secure enough for practical usage.

First we will concern ourselves with a network of only trusted peers. In this case Tinzenite should be very secure as long as the user ensures that no man-in-the-middle attack happens on setting up initial connections between new trusted peers. All further communications are encrypted and authorized by Tox. This in turn means that Tinzenite remains as secure as the software it is built on. As with any software all security is for naught should the user make an error or allow the stored data to leak outside of Tinzenite.

This is especially true as Tinzenite does not offer protection from other programs reading out the stored data in a clear text directory as this would make intended changes very difficult.

So what changes when encrypted peers are given access to the network? Not a lot for the communication between peers itself: Tox still theoretically guarantees a fully encrypted and authorized data channel. Since the data to be stored is encrypted on trusted peers before being sent, direct access to the user's data should also prove difficult, pending that the encryption algorithm is secure enough.

Since encrypted peers receive the model and all associated files only fully encrypted, this should not be a problem from a security standpoint. Trying to overwrite user data by replacing the binary blobs with random blobs is detected by trusted peers when comparing the unencrypted file with its hash from the unencrypted model: if they don't match the update can be marked as invalid and corresponding action taken. Detecting a wrong model is trivially done by decrypting it and checking if it is valid syntax.

Some meta information is leaked however. Since we encrypt on a per file basis it is theoretically possible to analyze file usage associated with size to identify pieces of the user's directory. This is only made easier by the fact that file sizes are roughly the same whether encrypted or not, with an accuracy down to the block size of the encryption scheme. For a future work proposal to solve this see section 6.2.2.

Possibly even more problematic is the clear text storage of the peer list even on encrypted peers as it can be used to determine the size of the user's Tinzenite network. To allow encrypted peers to facilitate file transfer between two mutually exclusively online peers, they must have access to this information. This problem is mitigated by the fact that Tox identifications are hard to guess and not shared beyond the Tinzenite peer network. Another possible risk is that Tox identifications can be used to monitor users' devices by querying the Tox network. Recovering a user's IP addresses should not be possible however as Tox employs onion routing to mitigate this risk [Toxc].

5

Implementation

In this chapter we will expand on the process of implementing our proof of concept. Before we can discuss the actual work of implementing the architecture we need to define the developing environment. This will include an introduction of the software libraries Tinzenite will base some of its functionality on.

5.1 Tools and Environment

In this section we will discuss the tools we used to build the software proof of concept. This includes libraries we utilized and the software used to write the programs.

5.1.1 Golang

As previously stated the proof of concept implementation will be developed using the programming language Golang. We will make full use of a range of features which we will briefly highlight in the following.

A Golang program will compile into a single native executable file, with statically linked dependencies. Cross compilation is available to all major operating systems and processor architectures. Unlike for example Java Golang does not depend on a virtual machine to run resulting in performance that is near to natively compiled C code. Furthermore Golang is not an object oriented language and based more on C than any other language. Golang is statically typed and garbage collected. This gives us type safety and removes the need to manage the memory ourselves. For a developer coming from Java a large

standard library helps to ease the transition: Golang offers such a standard library. We found writing Golang code to be less verbose than Java code for the same task while not being any harder to comprehend.

Golang handles errors and exceptions differently in comparison to Java. While in Java error handling is done explicitly via exceptions, Golang uses simple return values to signal errors. This poses less of a problem than it may initially seem to as Golang functions can return multiple values. Furthermore Golang exposes and allows working with object pointers. Concurrency is also directly built into the language via so called *"go routines"* and channels. Unlike Java which builds objects with class inheritance, Golang uses composition and interfaces to build objects. Building on this Golang does not even require the declaration of which interfaces an object implements – having the method of an interface means that that object implements the interface.

Golang also lacks a few features, notably generics and function overloading. We found these to be relatively easy to work around, although the lack of generics leads to an increase in redundant boilerplate code in some instances. A possibly higher hurdle is the lack of fine granular permissions: unlike Java Golang knows only private and public variables and functions, indicated by their names beginning with a lowercase or respectively uppercase letter.

As to the development environment surrounding Golang only a few specifics should be noted. A very nice feature is that packaging is directly built on top of version control systems. This means for example that *"github.com/xamino/tox-dynboot"* is both the package path and the URL where the package can be retrieved from. Within the code the package would be referenced by the name, commonly the last part of the package path. Golang also requires all code to be formatted according to its specifications which results in improved readability across different packages. A variety of tools are directly built into the development suite, including a tool to fetch packages from their path and a tool for vetting and formatting Golang code.

5.1.2 JSON

```
0 type Message struct {
1     Address string
2     Subject string `json:"omitempty"`
3     Content string `json:"Message"`
4     read    bool
5 }
```

```
0 {
1     "Address":"192.168.178.100",
2     "Message":"Log in successful!"
3 }
```

Listing 5.1: An example Golang struct with tags and its corresponding JSON representation. Note that *"Subject"* is missing from the JSON due to the *"omitempty"* tag and *"read"* due to it being private. Also note that *"Content"* has been renamed to *"Message"*.

Since the underlying Tox channel is built for text based messaging we propose to implement all peer to peer communication as a human readable messaging format. We will therefore utilize Javascript object notation, short JSON, as a machine readable message format while retaining easy readability for developers. As an added bonus Golang has support for converting objects by default thanks to the standard libraries. The generation of JSON can be fine tuned by utilizing in-language tagging. Listing 5.1 shows a very simple example.

5.1.3 Tox Binding

As stated in various instances before we will be building all peer to peer communication on the Tox core library [Gitd]. Since the library is implemented with the C programming language we require a Golang wrapper for it. Instead of implementing one ourselves which would have cost us a large amount of development time we chose to use an existing one. With some research we chose the wrapper written and provided by codedust via Github [gota].

At the start of the Tinzenite implementation this wrapper still lacked one significant feature that Tinzenite required, namely the capability of sending and receiving files. However a feature request [gotb] was submitted and promptly implemented by the maintainer. The maintainer was also forthcoming in helping us solve bugs and problems with our usage of the wrapper for which we are grateful.

5.1.4 Hadoop Client Binding

For the encrypted peer we required an implementation for a client for the Hadoop distributed file system. We chose the implementation by the Github user colinmarc. Notably we used the branch that adds write support [Gitb]. This library is not a wrapper but an implementation of a HDFS client written in Golang.

5.1.5 Environment

Tinzenite was implemented on the Arch Linux distribution Antergos [Ant], specifically the amd64 flavor. The Golang environment was set up using the corresponding Arch package [Arc]. We used the Golang tools provided by the package to compile our work. The code itself was written using the Atom text editor [Ato] with a variety of extensions, most notably the support extensions for Golang. Git [Gita] was used for the version control system, with a hosted repository on Github here [Gite].

5.2 Software Structure

Tinzenite was developed not as a single package but as a package collection where each package covers some part of the complete scope. In this section we will discuss the general layout of the packages and how they depend on one another. Figure 5.1 shows an informal diagram of the structure. All packages have their own repository on Github [Gite].

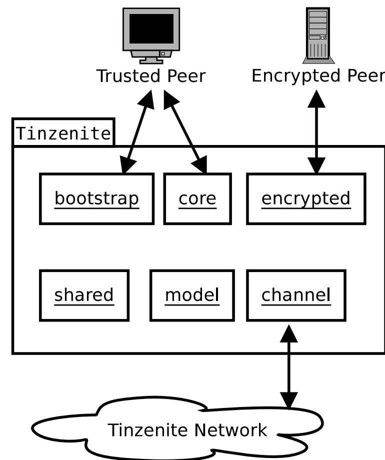

Figure 5.1: This diagram shows an informal representation of the package structure of our implementation.

bootstrap Contains the library for bootstrapping both encrypted and trusted peers to an existing Tinzenite network.

channel Building on the Tox wrapper implements an abstract object for all Tox related communication code.

core Implements the functionality for a trusted peer.

encrypted Implements the functionality for an encrypted peer.

model Contains the directory tracking code which manages a Tinzenite directory.

server Built on the *encrypted* package implements an example server program for an encrypted peer.

shared Contains various shared objects used by multiple packages.

tin Built on the *core* package implements an example user program for a trusted peer.

We began the implementation with the *model* and the *channel* packages, then built the *core* package on top of these. At one point we began writing shared code into the *shared* package for reuse. The *tin* package was added early on for testing and debugging purposes and grew alongside the trusted peer development. Once the basic functions worked we implemented the *bootstrap* package. Then we built the *encrypted* and *server* packages to implement the encrypted peer. Finally we adapted the *bootstrap* package for both peer types.

5.3 Highlights

The following section will serve to discuss highlights of the implementation. We will also expand on some aspects of the implemented functionality where we believe an expanded discussion is stimulating.

5.3.1 Model

The *model* package contains the model object used by the trusted peer to manage a tracked directory. Instances of the model can be created either by loading it from a JSON store or creating a new one. The model itself does not actively update itself if the underlying directory changes: instead an update must be triggered by the utilizing code. This allows the model to avoid having to employ file watchers. It thus falls to the utilizing code to call the update method in regular intervals to ensure that the model remains up to date.

The initial version of the model object only used file hashes to check for modifications, according to the Tinzenite specifications. For large files or a large amount of files this proved to be a very slow operation. Thus we also store the modification time as written to the file system upon creation and modification of a file to the model. This attribute is private to the model. As long as the modification time has not changed we do not need to recalculate the hash as nothing has changed since the last check. Thus the model is only required to recalculate the hash when the file actually has been modified. This greatly speeds up the update detection phase.

Apart from reacting to direct directory changes the model object also allows remote updates to be applied. Before a remote update can be applied a number of parameters must be checked – therefore the model object offers a `CheckMessage` method that returns whether the update must be truly applied. The update may be ignored for example if it has already been applied, or when it concerns an already completed removal as stated in section 4.2.3. Any call to `ApplyUpdateMessage` should be preceded by applying this message check. Note that we have moved most of the removal logic code to its own code file for easier comprehension.

Any application of an update may trigger a merge conflict. This is signaled by the model via an error. It is up to the caller to then handle the merge in a valid fashion, likely calling the model to apply resulting updates to the directory. Tinzenite handles merge conflicts as part of the *core* package.

Ignoring Objects

The *model* package also implements the matcher object. This object checks each directory for a *".tinignore"* file and if it exists applies it to any model work on the directory. The file containing these rules can be synchronized just as any other file at any level within a directory. Any new objects detected by Tinzenite that are listed in this file will not be created within the model, effectively keeping it out of the Tinzenite network. Since it may well happen that a *".tinignore"* is created at a later point and thus introduces an uncertainty in handling the files to be ignored we propose a simple solution: the file is only ever applied for object creations. Once a file has been created within the model it will be modified and deleted as any other file within the system, independent of whether it was or is to be ignored.

```
0 # DO NOT MODIFY!
1 /local
2 /temp
3 /receiving
4 /sending
```

Listing 5.2: The *".tinignore"* file contents for the *".tinzenite"* directory. Note that it allows comments. In this example only directories are excluded. Files can be excluded too: lines starting with the slash are considered file ignore rules.

The *".tinignore"* file is used in the *".tinzenite"* directory to control which parts are synchronized and which not, as mentioned in section 4.1. Listing 5.2 shows the contents of this *".tinignore"* file. Note that ignoring objects is supported within the complete Tinzenite directory: users that are so inclined can add such files themselves to control what objects Tinzenite synchronizes. We added support for comments to improve readability for users. The current implementation does not offer support for regular expressions or more advanced matching grammars, although these features should be comparatively simple to add.

5.3.2 Channel

The *channel* package wraps the Tox wrapper into the channel object via which all communication is sent. The channel object makes use of callbacks to allow callers to react to incoming messages, file transfers, and other events.

Since the underlying Tox instance must be called in regular intervals, the channel object implements a background go routine that keeps Tox ticking. To avoid locking up the Tox instance on callbacks, each callback is called asynchronously. If a callback can result in a direct response that must be passed to the Tox instance, another method is available.

Tox requires bootstrapping to known nodes to connect to the Tox network. This bootstrapping is different from the bootstrapping of peers we will discuss later. The channel object makes sure that the Tox instance is bootstrapped to the Tox network if it is not connected as long as the channel is kept running. We dynamically retrieve a list of available Tox nodes using a specially written package [Dyn]. This list is used to try to connect a given channel to the Tox network.

Unlike both the Tox instance and the Golang wrapper for it, the channel object wraps the file transmission nicely. If a file transfer request is received it asks the utilizing code via a callback whether to accept the transfer or not. If accepted it will again notify when the file has been successfully received or the transfer failed. Tox itself handles file transfers in data blocks. For our purposes the abstraction of this process greatly simplifies quite a lot of code that builds on file transfers.

The channel object furthermore offers a wide range of helpful methods that are not directly available from a Tox instance. This includes handling all addresses as hexadecimal encoded strings versus byte arrays and a number of status functions.

5.3.3 Tin Program

The *tin* package is an example implementation of a trusted peer built on the *core* package. While a true client built on a stable Tinzenite version should interface with the user via a

graphical user interface, we forewent this because of the increased development time. Thus the Tin program is a simple command line interface based program.

It offers a range of flags to make a Tinzenite directory easier to manage. Each instance of Tin should run for a single trusted Tinzenite peer. Additionally the program can also be used to create new peers and bootstrap them to the Tinzenite network. In the case of bootstrapping an encrypted peer the program will exit on successful completion. For trusted peers the program will immediately continue by running the directory. If a trusted peer receives a bootstrapping request it asks the user to validate the given peer address and requested trust level.

The program also automatically updates the local directory, requests remote updates, and synchronizes with available encrypted peers at certain intervals. This serves to prove that Tinzenite can run completely without user input. In fact apart from setting up a peer the synchronization runs fully without user input, even if merge conflicts arise. Care was taken to ensure that the entire program could run using as little resources as possible. Currently the program runs with negligible CPU and RAM usage even though synchronizing regularly. The only truly expensive operation where the program requires more performance is for hash generation and encryption when handling files. On an Intel Core i5-4440 with 4 physical cores at 3.1 GHz, each instance of Tin remains under 1% CPU and around 9 MB of RAM usage over a prolonged period of time when no update operations beyond model synchronization are executed.

The *core* package itself wraps the complete trusted peer code. Basically it uses the model object and the channel object to offer all trusted peer functionality. Notably it also implements how Tinzenite handles merge conflicts. For software written against it the core package exposes a Tinzenite object with which the trusted peer can be controlled.

5.3.4 Bootstrap

Initially we planned to include the bootstrapping code directly within the tin and core packages. However the differences in how a peer must react to incoming messages and transfers would have greatly increased the already substantial code complexity of these

packages. Therefore we moved all bootstrapping related code into its own package. This has the large advantage that programs can easily implement just bootstrapping if required to without needing any further package imports.

The basic function of bootstrapping is relatively simple. It requires the address of the trusted peer to connect to and sends a Tox friend request to it to initiate the connection. Once the friend request has been accepted and the trusted peer has been connected the actual bootstrapping takes place. For a new trusted peer that includes fetching the complete current state of the directory. Once the transfers are complete the bootstrapping code finishes and the directory can now be started as a normal trusted peer.

The bootstrap package was mainly implemented in one sitting and serves as a proof of concept that the previously implemented code for the trusted peer program could be reused easily. Apart from bug fixes we only had to update it once we implemented the encrypted peer functionality.

5.3.5 Server Program

Much like the Tin software the *server* package provides an implementation for the *encrypted* package. It offers a command line interface program for creating and running an encrypted peer.

```
0 package encrypted
1
2 type Storage interface {
3   Store(key string, data []byte) error
4   Retrieve(key string) ([]byte, error)
5   Remove(key string) error
6 }
```

Listing 5.3: The storage interface that the encrypted peer must implement to use the *encrypted* package. Comments have been removed.

Adding encrypted peers to Tinzenite required two tasks to be successful: the implementation of the encrypted peer and the extension of the trusted peer to be capable of utilizing it. The first task of implementing the encrypted peer proved to be comparatively simple. Basically all it has to do is offer a lock to any single trusted peer and the, when

locked, allow the fetching and retrieval of files. Most files the encrypted peer handles are user data files. One of the goals for this thesis was to support writing these files to the Hadoop distributed file system. Therefore we implemented the encrypted peer so that all user files are written with a specified storage interface as seen in listing 5.3. However this storage interface is not used for all files. The authentication file and the peer files are required to exist unencrypted and are required for the encrypted peer to run correctly. Therefore these files are written to disk within the *"org"* directory[1].

The storage interface allows encrypted clients, such as the *server* package provides an example of, to interface with any storage interface a developer requires it to. For this thesis we implemented two versions of the interface for the server program. First for debugging and personal use we implemented a version of the interface that simply writes all data to disk, named as the key of the data. Retrieval looks up the file by checking for the existence of a file named as the given key and returns the associated data. Then we implemented the actual Hadoop capable version of the interface. Our version of the interface creates a connection to a Hadoop distributed file system when initiating. All further file accesses will then be handled as operations on the Hadoop cluster.

Once the encrypted peer was implemented we turned to the more complex task of integrating encrypted peers into the capabilities of the trusted peer. This meant modifying the core package so that trusted peers could differentiate between trusted and encrypted peers and communicate with each accordingly. Upon receiving a message the trusted peer always first checks whether the message came from an encrypted peer or a trusted peer on a case by case basis. We do this by checking the address of the sender against the known peer list and ensuring that trusted peers have been authenticated. The message is then passed on to the specific logic for the type of peer.

The logic for handling an encrypted peer comes down to a relatively simple process. Upon successful locking an encrypted peer to itself, the trusted peer requests the encrypted model. If the encrypted peer responds with a notification that the file doesn't

[1] The encrypted peer does not share the directory structure of a trusted peer. The reason for this is that an encrypted peer lacks many of the features that went into the design of the *".tinzenite"* directory. Thus a simpler, flatter structure was chosen.

exist the encrypted peer is considered empty and the logic skips to uploading the current state of the trusted peer.

If a model is received the trusted peer must first compare the remote model against its own model to check for changes. Unknown updates of the encrypted peer are applied to the trusted peer by fetching the necessary files. Once this is complete the next step is to update the encrypted peer to the state of the trusted peer. Updates that the trusted peer has but the encrypted peer lacks are uploaded and finally the current model is encrypted and also uploaded. Then the encrypted peer is unlocked so that other trusted peers can access it too.

5.3.6 Golang Issues

We encountered a small amount of gotchas due to our unfamiliarity with the Golang language. In the interest of full disclosure we will use this section to briefly touch on these matters.

Tinzenite builds heavily on the standard libraries included with the language. Since Golang is still a relatively new language we expected to encounter bugs and issues as we implemented Tinzenite. However we only encountered a single bug. It effected the generation of a random integer for issuing a challenge. After creating a valid random `int64` value for the challenge we required a byte slice representation of the variable to encrypt it. We had to determine the size of the byte slice so that the variable could be stored in it. Research lead us to believe that utilizing the `byte.Size` method would allow us to make the byte slice just large enough to contain the random number. But the method always returned incorrect values. Our workaround is to simply create the slice large enough for all possible values at a small memory cost.

The good performance of Tinzenite was initially not the case. First versions of the trusted peer had a slow memory and processing leak resulting in steadily rising CPU and RAM usage until the host computer killed the process after a few minutes. We tracked this down to our usage of an endlessly running go routine utilizing `time.Tick` to run in intervals. As stated in the method documentation the underlying `Ticker` can not be

closed, thus resulting in a leak [Tic]. We worked around this issue by simply reusing the timing objects instead of recreating new ones within every interval.

Another problem was that we had no way to determine the type of `struct` we required to parse incoming JSON messages. Thus we implemented a *"Type"* attribute for all messages which we use to determine the correct type of the message. Golang's composition was not a viable way of solving this to our satisfaction since all message objects did not have any differing methods. The correct way to do this is to parse the incoming messages to an empty interface which is valid for all types. The type can then be checked and the message can then be casted.

```
0  type MsgType int
1
2  const (
3    MsgNone MsgType = iota
4    MsgUpdate
5    MsgRequest
6    MsgNotify
7    MsgLock
8    MsgPush
9    MsgChallenge
10 )
```

Listing 5.4: One of the enumerations we defined for Tinzenite. Note that for brevity we removed the comments.

For the defined messages that Tinzenite uses we heavily relied on enumerations for setting values. Coming from Java we expected to find similar `enum` functionality in Golang. This did not prove to be the case. Golang lacks a specific implementation of enumerations but does allow for something similar using the `const` and `iota` keywords for specified variable types. An example can be seen in listing 5.4. Our largest issue with this was that when converting these enumeration similes to JSON instead of the name of the enumeration the number value was output due to Golang seeing them as number value variables. Thus instead of having JSON where each enumeration was easily interpretable we had values such as `"MsgType":1` where what we wanted was `"MsgType":"update"`. The solution to this requires a lot of boilerplate code since Golang does not support generics at this point in time. For each defined enumeration type we had to write custom `MarshalJSON` and `UnmarshalJSON` methods, which we did where applicable.

5.4 Security

This section will discuss the scheme used to encrypt and decrypt files within Tinzenite and how the keys are stored and shared. Generally speaking Tinzenite uses two encryption layers to ensure file security. We will also briefly discuss the exact method we utilize for the challenge response algorithm.

The encryption scheme we used in Tinzenite is the NaCl library [BLS12] through its interoperable Golang package [Gold].From this package we utilize just three methods: `GenerateKey`, `Open`, and `Seal`. They are used to generate encryption keys, decrypt data, and encrypt data.

5.4.1 File Encryption

Every Tinzenite network generates a single permanent key pair which is used to encrypt and decrypt data. Each peer upon creation generates these keys but will overwrite them with the network's keys if connected to an existing network during bootstrapping to ensure that each Tinzenite peer uses the same keys. Otherwise peers could not share encrypted peers between themselves as each would use a different set of encryption keys. The keys themselves are generated using the cryptographically secure random number generation provided by the host operating system to ensure that they are truly random.

File data is encrypted before sending it and decrypted after receiving it when exchanging data with encrypted peers. To encrypt and decrypt data via NaCl we require an associated nonce which must be atomic but unique to every encryption decryption cycle. We require a way to store and retrieve the nonce for each encrypted data blob. Instead of writing the nonce somewhere else we opted to prepend the nonce to the encrypted data. Thus the first 24 bytes of every encrypted file contain the nonce required to decrypt it again. Figure 5.2 shows an example of how this works.

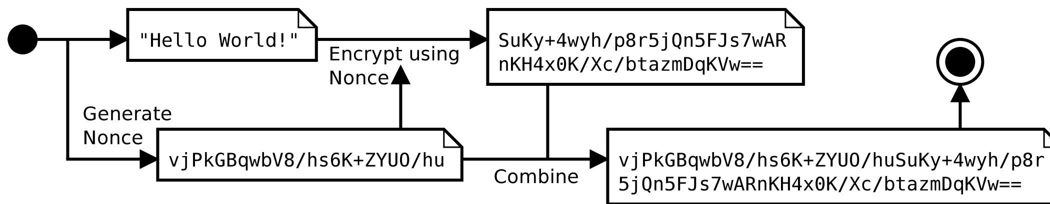

Figure 5.2: This diagram shows an example of how the nonce is integrated into the data for an encrypted message.

When decrypting data we must thus remove the nonce from the data before decrypting it. Since the decryption requires the nonce we can simply use it directly. The nonce is then discarded and the decrypted data is ready to use.

5.4.2 Key Encryption

Now that all data can be encrypted and decrypted correctly the question turns to how to store the corresponding keys safely. At first glance a simple solution would be to synchronize the keys only between trusted peers. However this would not allow an implementation for a web peer as described in section 3.4.4. A temporary peer could not request the keys because it could not authenticate itself. Thus the keys must be stored alongside the encrypted data.

It is immediately obvious that simply storing the keys in plain text next to the encrypted data offers no security. So the keys must be encrypted too but with another set of keys. The question then is where to store these keys for encrypting the encryption keys. Since Tinzenite should be user friendly and secure we opted for an interesting algorithm for generating the keys that unlock the actual encryption keys. To differentiate the two different sets of keys we will reference the encryption keys which are used to encrypt and decrypt the other keys as the box keys.

On creating a Tinzenite network the user is asked for a password or better yet a passphrase [Mun11]. We then use this passphrase to generate the box keys used to encrypt the encryption keys. Now the encryption keys can be encrypted with the box keys and safely transmitted to both trusted and encrypted peers. The question now arises how other trusted peers can retrieve the box keys.

The answer lies in the user given passphrase and the way the box keys are generated. Every trusted peer must be unlocked by the user with the passphrase[2] to decrypt the encrypted peers. This passphrase is not stored somewhere in the Tinzenite network for a directory, meaning an attacker has to guess it.

```
0  func (a *Authentication) convertPassword(password string) (public *[32]byte, private
       *[32]byte, err error) {
1    hasher := fnv.New64a()
2    hasher.Write([]byte(password))
3    seed := int64(hasher.Sum64())
4    seededRandom := unsecure.New(unsecure.NewSource(seed))
5    wrapper := staticRandom{random: seededRandom}
6    public, private, err = box.GenerateKey(wrapper)
7    if err != nil {
8      return nil, nil, err
9    }
10   return public, private, nil
11 }
```

Listing 5.5: The method for generating the box keys from a given passphrase. The `staticRandom` object is a wrapper for the random source so that it can be passed to the `box.GenerateKey` as the correct type. It is defined elsewhere.

The box keys are then generated from the passphrase. First we hash the passphrase to a number hash value. This number hash is then used to seed a pseudo random number generator. We then utilize this deterministic random number generator to create the box encryption keys. Listing 5.5 shows the entire code required to generate the box keys from a given passphrase.

This method of storing the encryption keys has a few advantages over other approaches. An important advantage of decoupling the passphrase from the encryption keys via the box keys is that this allows the user to change his passphrase if desired. Tinzenite must then only decrypt the encryption keys with the old passphrase and encrypt them with the new one. The update will then propagate through the network[3]. Encrypted peers do not need to be encrypted anew since the actual encryption keys remain the same. Another advantage is that the entire encryption process is tied to a passphrase that is simple to

[2]Note that the trusted peer should store the passphrase once the user has entered it to avoid needless repetition and to increase the ease of use for future accesses.

[3]Note that we make no provisions for invalidating old passphrases since this would require substantially more encryption work and would most likely not be enforceable.

remember and the user can freely assign and modify. Apart from said passphrase the user must do nothing to ensure full encryption security for the Tinzenite network.

5.4.3 Challenge Response

The challenge response we use in Tinzenite is built on a very simple challenge. For each challenge we generate a random number and locally store it, then encrypt it with the data encryption keys and send it to the other peer. This responding peer decrypts the message and retrieves the number. It then increments it by one, encrypts its answer, and sends it back[4]. If the received number is one value higher than the stored number, the challenge response is valid.

The challenging peer can thus be satisfied that the other peer is valid because it proved that it could validly decrypt and encrypt the correct values. The responding peer knows the challenging peer is authenticated because it could issue a valid challenge for the network data encryption keys. In all the challenge response mechanism for Tinzenite requires only two messages and a single random number to validate both sides of the exchange.

[4]What we actually do with the number is irrelevant as long as both sides of the challenge response algorithm use the same operation. The only property the operation must have is that it is sufficient to prove that both sides could read the unencrypted number and work on it.

6

Results and Recapitulation

This chapter will discuss the proof of concept implementation and recapitulate on how the architecture was implemented. Specifically we will elaborate on the state of our implementation in relation to existing services. We will then discuss future work. Our future work can be divided into two broad categories: improvements on the existing solution and expansion on top of it.

6.1 Comparison

In this section we will briefly and informally compare Tinzenite with existing solutions. Note that an actual study comparing our solution with other storage providers was not the intended goal of this thesis. All information used to compare the different solutions is to be taken with a grain of salt.

6.1.1 Network Performance

Tinzenite shares a large advantage with the other related peer to peer solutions that have been discussed. Because the peers communicate directly with each other the synchronization speed for peers within the same local area network is substantially higher than the server-client solutions that must first upload data to a remote server. In the local area network where Tinzenite was mainly developed the only speed limit was that of the wireless internet for data transfers.

When transferring data between two peers over the internet at large Tinzenite is restricted by the broadband upload speed of the user's internet connection. While generally

speaking the upload speed is only a fraction of the download speed for most consumer connections, Tinzenite does not suffer under this. This due to that when comparing transferring a file via Tinzenite and any server hosted service, both require an initial upload of the file. Tinzenite actually wins in terms of speed because by the time the upload has been completed, the other Tinzenite peer already has the complete file downloaded. Server-client alternatives must still download the file to the other client additionally.

6.1.2 Usability

Here Tinzenite in its current state definitely loses out to existing solutions. While running the software requires little to no user interaction, the setup and management of peers is currently only available via command line interface programs. Nothing speaks against implementing a graphical user interface in principle. This was not an essential part of our work however so no such client was implemented due to time constraints. And while Tinzenite theoretically supports web access to encrypted peers (see section 6.2.2 below), this capability was also not tested or implemented. We believe that web access would improve usability by a large margin and allow Tinzenite to close the feature gap.

Tinzenite's features are complete in that it can and does synchronize directories correctly. However further work should be done for edge cases where the usability could be improved, such as for merge conflicts. By notifying and offering support in resolving merge conflicts Tinzenite could provide a more active support for its usability. Such features could easily be integrated into a client program without requiring modifications to the underlying Tinzenite code.

A feature important for the security of a Tinzenite network is support of clients for strong passphrases. Users should be guided to create long and secure passphrases because then they become increasingly harder to guess.

6.1.3 Security

Tinzenite's security is built on one of the most scrutinized encryption libraries currently in existence, the NaCl library [Ber09]. Therefore the encrypted data that untrusted, encrypted peers receive should be more than sufficiently secure to deny unauthorized third party access even if the server peer is accessible. Should services running encrypted peers be requested to hand over user data by hostile governments, said data would be inaccessible without guessing the passphrase[1]. Thus the security of the user data primarily depends on the user's willingness to use a sufficiently complex passphrase. Note that Tinzenite does not provide data deniability: if the user is forced to give up the passphrase [Mun09], no further mechanisms prevent full data access.

Theoretically the capability of working with encrypted peers is Tinzenite's distinguishing point between all existing implementation mentioned in section 2.1. Like other peer to peer solutions Tinzenite works directly between peers with most of the associated advantages. Additionally Tinzenite allows for off site backup of the users' data. Unlike the server-client solutions Tinzenite does not require an account or trust in a third party. We believe that Tinzenite therefore combines the best of both worlds in regards to security while retaining readily available access to user stored data for authorized entities.

None the less, Tinzenite should not be trusted with secure data at this point in development. Since encryption is hard to implement correctly we are not confident enough of our implementation of the various aspects of the encryption scheme. Furthermore Tinzenite should only be trusted after a security audit has been performed on the core logic. Tinzenite's open source nature however allows any interested party to audit the code at their own leisure. Tinzenite does leak some information: namely the peer list is available unencrypted along with the authentication file. While a small risk as no personal user information is stored in those files it is still a possible attack vector for a malicious peer.

[1] Without knowing the passphrase the box keys can not easily be determined. Therefore the actual file encryption keys which are encrypted with the box keys are stored securely enough to ensure confidentiality.

6.2 Future Work

Tinzenite offers a lot of room for future work, both concerning the current implementation and expanding on the provided work. This section will serve to discuss many of the points we believe could greatly benefit any further work on Tinzenite. Thus we will begin this section by discussing changes to the current Tinzenite architecture and implementation. We will conclude this section by offering an outlook of further work that could be based on the existing implementation.

6.2.1 Improvements to Existing Work

Since our implementation of the architecture was built from the ground up without previous knowledge of many of the aspects touched on by this work, we encountered a number of things that should and could be improved.

Data Transfer

We believe our implementation not to be fully optimized for the best possible data transfer characteristics between a peer network. Thus improvements to how Tinzenite fetches and sends data could be implemented to improve robustness and speed of data transfers.

In our implementation when a trusted peer receives an update, the associated file is fetched from the peer where the update originated from. If the update is received from multiple peers only the first peer is used to fetch the data. Building on the same ideas that led to the development of the BitTorrent protocol, we could implement swarm fetching capabilities. This would allow the unchoking of saturated peers where upload speeds are slow and allow the file request to complete even if one peer out of many goes offline or encounters issues.

Issues may arise for implementing this for two reasons. If delta fetching is implemented as described below, care has to be taken to keep the swarm behavior compatible even if multiple peers have varying states of the original file. Another source of possible

issues is how to combine it with the existing encrypted peer behavior. Since encrypted peers are currently locked to a single trusted peer for a synchronization this precludes having them partake in a swarm apart from the issue that the encrypted peer will transfer encrypted data while trusted peers will transfer unencrypted data.

Peer Behavior

Trusted peers of the current Tinzenite implementation synchronize with timing based intervals. While this works sufficiently for the proof of concept, ideally it would not be the case. Instead peers should dynamically adjust the timing and order of operations for when to synchronize based on the network environment and current status of the own directory.

An example for this includes pausing the remote synchronization interval if outside changes are still being fetched to avoid unnecessary double fetch requests and associated work. Another example would be extending the Tinzenite *core* package to allow for finer control over which peers will synchronize and then implementing the client program so that peer synchronizations happen in a smarter fashion. To reduce the work load for programs building on Tinzenite this could even be implemented within Tinzenite itself. This could include synchronizing only once when initially connected and then simply updating locally, avoiding unnecessary complete model comparisons. Synchronization with encrypted peers could also be improved to avoid locking multiple encrypted peers at once and ensure that they are kept up to date at a reasonable rate.

Delta Updates

Fetching a file in Tinzenite currently always transfers the complete binary data, even if only a small part of the file was changed between two versions. Thus an improvement would be to implement that Tinzenite only sends the differences between two versions of a file between peers. We propose to use rsync algorithm for this [TM+96], specifically the librsync implementation [Lib]. The required information for the library to work can be integrated into the existing request messages. Delta updates could only be used

between trusted peers since the encrypted data is completely different for every change. Only needing to send the changed part of a file should increase the speed of Tinzenite transfers immensely, expecially for large files.

Server Peer

The current implementation for the encrypted peer was implemented for a single Tinzenite network instance. It could be extended to provide service for multiple Tinzenite networks and multiple users. User accounts can be differentiated by reading parts of the authentication file: the user name can be checked with the bcrypt hash. Care should be taken to ensure that the user does not provide the same access password like the passphrase used to access the Tinzenite encryption, although this won't be enforceable by Tinzenite.

Another feature that the server client should be capable of is enforcing potential size restrictions. This feature may also be used for a future mobile client as described in section 6.2.2. What this means is that Tinzenite should support clients refusing to fetch additional files to enforce a specified size of a directory. It could be implemented by building on the shadow files capability which we will expand on in the next section. The interesting case is of course what happens to files that are above the limit after a modification: we propose either making the file a shadow file as soon as it crosses the limit or allowing modifications to push the size above the limit temporarily. For encrypted peers the enforcement of size restrictions must be handled by the trusted peers. This in turn means that encrypted peers must be capable of denying additional updates since trusted peers can not be trusted to correctly enforce a size limitation.

Shadow Objects

Depending on the location of a client a user may wish to only access specific files without having to get an entire set or updates. This is a nice feature to have in the case of space and bandwidth restricted devices such as mobile devices or for the web interface. This feature could also be used to prioritize which objects Tinzenite will fetch first.

Functionality for the shadow file feature is available via the currently unused *"shadow"* attribute. It affects only files directly as the creation of directories is not significant from a size point of view. The attribute only serves as a shortcut to set all files of a directory implicitly to being shadowed. If files are marked as shadow files they are not updated on the disk, only their model. By setting the shadow flag to true the client will then immediately try to fetch the binary file from connected and available peers.

Shadow files pose a few additional difficulties that would have to be solved. First and most trivial: what happens to an already synchronized file when the shadow attribute is set? We propose that the file is immediately removed although this could be expensive in terms of bandwidth if the users quickly change their mind again because the file must then be fetched again all anew. A more sophisticated approach would integrate the size restriction capability of the client as proposed previously. By setting the space limit to a number below the full size of the directory, files will only be immediately removed if near the space limit. If the users change their mind the file may thus still be immediately available.

So what do peers do if they receive a model update where the shadow flag is set? It is important to note here that the shadow flag is considered to be transient when synchronizing models, meaning its value is considered to be local only. However it is still sent because it is used to determine whether the receiving peer can fetch an update from the other peer if applicable. Again it is up to the peer what happens upon receiving a shadow file update: normally a peer that has a non shadow copy of the file will ignore shadow updates as it can not fetch the binary file update successfully from it. It will then have to wait for another peer to offer the update where the attribute is not set.

The final edge case is a challenging one: what Tinzenite does not provide is a way to ensure that one full copy of the shadowed file is always kept somewhere. If the user marks a file as shadowed on all peers it may well happen that Tinzenite loses the file. For now we propose to avoid this by explicitly warning the user of this possibility. One way to mitigate this risk is by allowing user defined shadowed files only for specific clients: we can probably assume that any full desktop peer should always retain a full copy of the directory anyway.

Implementation Improvements

Our current implementation, while working, is surely not the best way to implement it. Not all possible error cases are handled in the most optimal way possible. Furthermore the implementation of Tinzenite could use extensive testing and debugging.

6.2.2 Expanding Work

Tinzenite offers a lot of room to build on. In this section we will discuss some proposals for building on our work instead of modifying it. Note that some expanded features require features from the previous section.

Encrypted Peer Synchronization

Each encrypted peer is a single point of back up in the current state of our work. This requires each trusted peer to check up on each encrypted peer in turn and ensure that all of them are kept at the same synchronization state. This could be improved by allowing encrypted peers to update each other. Care has to be taken that encrypted peers will not cause merge conflicts that they can not resolve, since both the model and its objects are fully encrypted. We thus propose to extend the Tinzenite architecture for all encrypted peers to work as a single meta peer, where in truth multiple instances are kept in the same, most up to date state possible.

This would require some further logic in how to detect and merge different encrypted peer states. Generally it could be done if the directory model had an unencrypted version state attached to it. As long as no model conflicts arise, all encrypted peers can keep updating themselves to the most current version. If however two encrypted peers receive conflicting update states, the entire swarm of encrypted peers would need to wait for a trusted peer to resolve the issue. This would require some large extensions to the current protocol, but in theory should be doable.

Data Obfuscation

In the case of encrypted peers simply encrypting a file may not be sufficient to prevent meta information collection on the directory contents. Thus we propose that future work could include modifying the encrypted peer implementation and the transfer protocol so that trusted peers send not only encrypted but also obfuscated data to encrypted peers, effectively implementing an oblivious storage system [GO96]. This especially makes sense if combined with the swarm behavior mentioned previously and the encrypted peer synchronization. The entire group of encrypted peers could then be used to obfuscate and store data redundantly. This would increase the third party security and further reduce the trust of said party required. Obfuscation would be sufficient to hide most meta data that may be deduced from encrypted synchronization.

Update Feedback

While our implementation allows for a basic feedback of large file transfers in the form of basic progress meters, a lot more detailed and better specified feedback could be offered to peers. This would allow them to signal to the user of outstanding updates – such as updates that it has received but not yet applied because the transfer is pending. Generally such features would also tie in with the better control of peer behavior previously mentioned. If the user can quickly see at a glance the general state of the local trusted peer or even other connected peers, usability of the entire Tinzenite network increases. It would also increase the user's trust that the synchronization is working as intended.

Additional Peer Versions

Our current implementation of Tinzenite offers two peers: a standard trusted peer for a personal computer and a peer for encrypted storage. As discussed in section 3.4 we have already considered adding a mobile client for smartphones and a web interface client for temporary access to an encrypted peer, plus a passive peer for secure cold storage.

A mobile peer would be a trusted active peer of the Tinzenite network. As previously discussed however the mobile peer would most likely not retain a full copy of the Tinzenite directory due to size and bandwidth limitations. Thus both shadow objects and size restrictions are likely prerequisites for a mobile client. On the positive side little else would need to be changed in our provided work as Golang can be executed on both of the most popular mobile operating systems currently in use, Android and iOS. Indeed the entire application could be written in Golang, building on our already completed work, by utilizing the *mobile* package [Golb].

The web interface peer would be an interesting challenge. It would allow the users to log in to a web server and enter their passphrase. The web peer would then be capable of fetching and decrypting the model file and allow the user to upload and download encrypted data directly from an encrypted peer. This could happen entirely without requiring the underlying Tox communication architecture. All data to and from the web server would be fully encrypted since decrypted data would only be kept on the web interface peer. The moment a user closes the web page all temporary data would be discarded. This would enable users to access their data stored in the Tinzenite network anywhere where they have internet access, as long as they have encrypted peers that enable this feature.

Finally a cold storage peer could be offered. Technically it would not be a true peer of the Tinzenite network, but for the sake of this text we will reference it as a passive peer. A user could command a trusted Tinzenite peer to utilize a storage location as a passive peer. Tinzenite would then encrypt all local files and its current state and write the data to the specified location. This location could be anything from a USB stick to more permanent storage device. If the users wish to update the passive peer they would not even have to do it at the same peer: any other trusted peer could be used too. This other trusted peer would, similar to how the encrypted peer works, read and synchronize the passive peer against itself. This would allow passive peers to be used both as safeguards against data loss and even as secure transport containers. Two trusted peers not connected via the internet could be kept synchronized manually by regularly moving a passive peer between them.

File Versioning

Another advanced feature that would be very nice to have and close the gap of feature parity between Tinzenite and other existing services would be the capability to offer file versions built directly into the Tinzenite architecture. Indeed the core protocol would not even have to be changed to support this: all that is required is the capability to keep old files for a specific time somewhere where the peer can reinstate them if the user wishes to.

We propose to implement this as follows. First, for every created object that Tinzenite detects, it copies the initial version into a hidden directory for future reference. Then, whenever a change within the file is detected, the delta update functionality can be used to store the difference between the old version and the new version of the file. A file removal could be marked specifically. A setting could be introduced to allow the user to control how far back different versions of an object are kept to keep storage requirements at a reasonable level. Thus it should be comparatively easy to only retain the last three versions of an object. Trusted peers could then offer assistance in actually managing the versions if the user wants to reinstate one over the current object.

7 Conclusion

This chapter serves as the conclusion of our work, both the theoretical part and the practical proof of implementation. We will terminate this thesis with a closing statement.

7.1 Theoretical Work

We have developed and discussed the design decisions for a peer to peer encrypted file synchronization software. Before defining our concept we looked at existing work and academic papers to help us define the features we would like to cover in chapter 2. In chapter 3 we discussed the general scope of the undertaking and defined the concept of this work. We combined most features users have come to expect from any other available data storage service with features required to retain the security of the users' data into an encompassing concept. Chapter 4 thus served to define the actual architecture we would implement as a proof of the concept of the basic functionality. We discussed how we proposed to define the storage space to work for all required use cases and how the peers would communicate between themselves. We then discussed our implementation of the proof of concept work in chapter 5. We went into detail in how our implementation is structured and how we implemented core aspects of the proposed architecture. Finally we wrote about the finished implementation in chapter 6. We explained the current feature state of the implementation in comparison to existing work. Furthermore an outlook on the multiple possible future improvements and expansions was given.

7.2 Implementation Work

A large part of our work was the implementation of a proof of concept for the proposed Tinzenite software suite. This includes example user programs for both the trusted peer and the encrypted peer. All of our code can be found in the Github organization for Tinzenite [Gite].

Beyond providing a proof of concept implementation which covers the essential scope, we also used tools, a programming language, and libraries new to us for our work on Tinzenite. Thus we expanded on their implications in section 5.1. Our implementation language of choice was Golang. We built the communication of Tinzenite on the Tox communication protocol, thus allowing Tinzenite to build on an existing end to end encrypted communication standard. For the encrypted peer we implemented a storage interface that allows server clients to store the user data in any available storage system. We included two example implementations for this interface: a simple one that writes the data to disk and another one that allows the server client to write user data to the Hadoop distributed file system.

7.3 Closing Statement

We have shown our results of developing and implementing a data synchronization library for secure peer to peer data storage. This includes preparatory work not only by comparing existing services and academic papers but also developing a new protocol for data exchange. This library, named Tinzenite, was subsequently implemented as a proof of concept and two example client implementations were also developed, a trusted and an encrypted peer. While various aspects could still be improved and expanded on, the basic scope of the thesis has been completed.

Retrospectively we are satisfied with the promise of Tinzenite. We consider our work to be usable for academic, exploratory, and developing purposes but would refrain from utilizing it in a real world scenario due to outstanding security and improvement considerations.

8 Glossary

Encrypted Connection

An encrypted connection is utilized between encrypted peers and between an encrypted peer and a trusted peer. Messages are partially modified (encrypted and / or anonymized) to preserve the data privacy.

Encrypted Peer

An encrypted peer contains an encrypted copy of the directory. The keys for encryption are not shared with it. All messages it receives contain anonymized or encrypted information (for example file names) wherever necessary.

Fourth Party

We define fourth parties for the purpose of this work as follows: for a first party communicating with a second party using a service provided by a third party, fourth parties are those that partake in this communication without the first or second party knowing of it. Specifically this includes governmental agencies such as the National Security Association or hostile entities such as industrial espionage.

Nonce

A nonce is an arbitrary number that should be used only once. Nonces play important roles in cryptographic practices. In Tinzenite nonces mostly serve to avoid replay attacks by making each encryption unique even if the same content is reencrypted.

Third Party

A third party is defined as a party that offers a service for the first party and second party to communicate. This can range from a messaging service such as Skype to a service provider such as Dropbox.

Tinzenite

The name of the core software library of this thesis.

Trusted Connection

A trusted connection is only between two trusted peers. All information that flows over it is complete and in clear text.

Trusted Peer

A trusted peer has access to the encryption keys and contains an unencrypted copy of the directory.

List of Listings

4.1 Authentication JSON Object 29
4.2 Peer JSON Object 30
4.3 Version JSON 33
4.4 Directory JSON Model 34
4.5 File JSON Model 35
4.6 Authentication Message 47
4.7 Request Message 48
4.8 Update Object Message 49
4.9 Lock Peer Message 50
4.10 Push Message 50
4.11 Notify Message 51

5.1 Golang JSON Example 59
5.2 Meta Ignore File 63
5.3 Encrypted Storage Interface 66
5.4 Golang Enum Example 69
5.5 Golang Box Key Generation 72

List of Figures

1.1 Tinzenite Icon 4
1.2 Thesis Structure Diagram 5

2.1 Example Network Structures 7

3.1 Tinzenite Software Diagram 23

4.1 Meta Folder Structure 28
4.2 Removed Folder Structure 31
4.3 Data Model Example Structure 33
4.4 Object State Diagram 37
4.5 Connection State Diagram 42
4.6 New Connection Sequence Diagram 44

5.1 Tinzenite Structure 61
5.2 Encryption Example 71

List of Tables

2.1 Existing Software Comparison 12

4.1 Peer Object Version States 54

Bibliography

[And] Android. http://www.android.com/. Accessed 2015.05.29.

[Ant] Antergos | Your Linux. Always Fresh. Never Frozen. https://antergos.com/. Accessed 2015.10.13.

[Arc] Arch Linux - go 2:1.5.1-2 (x86_64). https://www.archlinux.org/packages/community/x86_64/go/. Accessed 2015.10.13.

[Ato] Atom. https://atom.io/. Accessed 2015.10.13.

[Ber09] Daniel J Bernstein. Cryptography in nacl. *Networking and Cryptography library*, 2009.

[Bita] BitTorrent.org. http://www.bittorrent.org/index.html. Accessed 2015.10.12.

[Bitb] BitTorrent Sync. https://www.getsync.com/. Accessed 2015.04.24.

[BLS12] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *Progress in Cryptology–LATINCRYPT 2012*, pages 159–176. Springer, 2012.

[Bor07] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

[Bor14] Andrew Bortz. We've got your back | Dropbox Blog. https://blogs.dropbox.com/dropbox/2014/06/weve-got-your-back/, June 2014. Accessed 2015.05.14.

[Box] Boxcryptor | Encryption for cloud storage | Window, Mac, Android, iOS | boxcryptor.com. https://www.boxcryptor.com/en. Accessed 2015.04.24.

[BP98] Sundar Balasubramaniam and Benjamin C Pierce. What is a file synchronizer? In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 98–108. ACM, 1998.

[Dro] Dropbox. https://www.dropbox.com/. Accessed 2015.04.24.

[Dyn] xamino/tox-dynboot. https://github.com/xamino/tox-dynboot. Accessed 2015.10.12.

[Gita] Git. http://www.git-scm.com/. Accessed 2015.05.26.

[Gitb] colinmarc/hdfs at write-support. https://github.com/colinmarc/hdfs/tree/write-support. Accessed 2015.10.13.

[Gitc] gopherjs/gopherjs. https://github.com/gopherjs/gopherjs. Accessed 2015.10.28.

[Gitd] irungentoo/toxcore. https://github.com/irungentoo/toxcore. Accessed 2015.10.12.

[Gite] Tinzenite. https://github.com/tinzenite. Accessed 2015.10.17.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[Gola] Frequently Asked Questions (FAQ) - The Go Programming Language. http://golang.org/doc/faq. Accessed 2015.05.29.

[Golb] mobile - GoDoc. https://godoc.org/golang.org/x/mobile. Accessed 2015.10.17.

[Golc] The Go Programming Language. https://golang.org/. Accessed 2015.05.29.

[Gold] box - GoDoc. https://godoc.org/golang.org/x/crypto/nacl/box. Accessed 2015.10.11.

[Goo] Google Drive - Cloud Storage & File Backup for Photos, Docs & More. https://www.google.com/drive/. Accessed 2015.04.24.

[gota] codedust/go-tox. https://github.com/codedust/go-tox. Accessed 2015.05.29.

[gotb] Implement File Send · Issue #3 · codedust/go-tox. https://github.com/codedust/go-tox/issues/3. Accessed 2015.10.12.

[GP13] Barton Gellman and Laura Poitras. U.S., British intelligence mining data from nine U.S. Internet companies in broad secret program - The Washington Post. http://www.washingtonpost.com/investigations/us-intelligence-mining-data-from-nine-us-internet-companies-in-broad-secret-program/2013/06/06/3a0c0da8-cebf-11e2-8845-d970ccb04497_story.html, June 2013. Accessed 2015.10.16.

[Gri14] Erin Griffith. Who's winning the consumer cloud storage wars? - Fortune. `http://fortune.com/2014/11/06/dropbox-google-drive-microsoft-onedrive/`, November 2014. Accessed 2015.10.05.

[Had] Welcome to Apache™ Hadoop®! `https://hadoop.apache.org/`. Accessed 2015.05.29.

[Hol14] Martin Holland. NSA-Skandal: Provider hilft BND angeblich beim Zugriff am Internet-Knoten DE-CIX | heise online. `http://www.heise.de/newsticker/meldung/NSA-Skandal-Provider-hilft-BND-angeblich-beim-Zugriff-am-Internet-Knoten-DE-CIX-2261805.html`, July 2014. Accessed 2015.10.05.

[Lib] librsync. `http://librsync.sourcefrog.net/`. Accessed 2015.05.29.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[Mun09] Randall Munroe. xkcd: Security. `https://xkcd.com/538/`, February 2009. Accessed 2015.10.28.

[Mun11] Randall Munroe. xkcd: Password Strength. `https://xkcd.com/936/`, August 2011. Accessed 2015.10.15.

[One] Microsoft OneDrive. `https://onedrive.live.com/about/en-us/`. Accessed 2015.06.03.

[Pik12] Rob Pike. Go at google: Language design in the service of software engineering. `https://talks.golang.org/2012/splash.article`, October 2012. Accessed 2015.10.14.

[PJGH+98] Thomas W Page Jr, Richard G Guy, John S Heidemann, David H Ratner, Peter L Reiher, Ashish Goel, Geoffrey H Kuenning, and Gerald J Popek. Perspectives on optimistically replicated, peer-to-peer filing. *Software-Practice and Experience*, 28(2):155–180, 1998.

[PM99] Niels Provos and David Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.

[RC+01] Norman Ramsey, El Csirmaz, et al. An algebraic approach to file synchronization. *ACM SIGSOFT Software Engineering Notes*, 26(5):175–185, 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. *A scalable content-addressable network*, volume 31. ACM, 2001.

[RPG+96] Peter Reiher, Jerry Popek, Michial Gunter, John Salomone, and David Ratner. Peer-to-peer reconciliation based replication for mobile computers. In *European Conference on Object Oriented Programming, Second Workshop on Mobility and Replication*. Citeseer, 1996.

[RT 13] Snowden leak: NSA secretly accessed Yahoo, Google data centers to collect information — RT USA. http://rt.com/usa/nsa-secretly-access-yahoo-google-982/, October 2013. Accessed 2015.05.14.

[RT 14] 'Hostile to privacy': Snowden urges internet users to get rid of Dropbox – RT News. http://rt.com/news/195244-snowden-rid-dropbox-privacy/, October 2014. Accessed 2015.05.14.

[Syn] Syncthing. https://syncthing.net/. Accessed 2015.04.24.

[Tic] time - The Go Programming Language. https://golang.org/pkg/time/#Tick. Accessed 2015.10.14.

[TM+96] Andrew Tridgell, Paul Mackerras, et al. The rsync algorithm. *Made available in DSpace on 2011-01-05T08: 37: 42Z (GMT).*, 1996.

[Toxa] Tox. https://tox.chat/. Accessed 2015.10.05.

[Toxb] toxcore/Crypto.md at master · irungentoo/toxcore. https://github.com/irungentoo/toxcore/blob/master/docs/updates/Crypto.md. Accessed 2015.05.29.

[Toxc] users:techfaq - Tox Wiki. https://wiki.tox.chat/users/techfaq?s[]=onion#what_is_stopping_people_from_tracking_me_through_the_public_dht. Accessed 2015.10.11.

[Whi] Open Whisper Systems » Home. https://whispersystems.org/. Accessed 2015.10.05.

Name: Tamino P.S.M. Hartmann Matriculation Number: 722891

Affidavit

I herewith declare in lieu of oath that I have composed this thesis without any inadmissible help of a third party and without the use of aids other than those listed. The data and concepts that have been taken directly or indirectly from other sources have been acknowledged and referenced. This thesis has not been submitted, wholly or substantially, neither in this country nor abroad for another degree at any university or institute.

Ulm, the ....

Tamino P.S.M. Hartmann