The Delta File Transfer Protocol

A Distributed File System for Local Networks

Project Report
SW704e20

Aalborg University
Computer Science

AALBORG UNIVERSITY
STUDENT REPORT

Computer Science
Aalborg University
http://www.aau.dk

Title:
The Delta File Transfer Protocol

Theme:
Internet of Things

Project Period:
Fall Semester 2020

Project Group:
SW704e20

Participant(s):
Christian Thomas Hurley
Christoffer Hansen
Fredrik De Frène
Mads-Bo Bomholt Lassen
Mathias Andresen
Simon Park Kærgaard

Supervisor(s):
Peter Gjøl Jensen

Copies: 1

Page Numbers: 71

Date of Completion:
December 21, 2020

Abstract:

Motivated by the UN's sustainable development goals and a lack of user transparency in cloud storage, we analyze how we can improve a person's digital storage of private data beyond cloud storage solutions. We present an analysis on how synchronizing files between a person's devices can be done by utilizing their unused storage. We believe this can have positive effects regarding how electricity usage and hardware production influence the environment. We present our design of a distributed file-sharing system for local networks, which is implemented as a fully-fledged Windows application. We use Node.js and its extensive selection of packages to implement the application and utilize a difference-algorithm to calculate the delta between two versions of a file. We conduct tests of the system's features, states, and performance. In our use-case specific performance test, using three devices, we outperform DROPBOX by ~0.34 seconds and GOOGLE DRIVE by ~8.68 seconds but perform worse than DROPBOX with LAN sync by ~0.22 seconds. We conclude with a summarizing view of our system: The system provides a way to utilize unused storage resources by letting a user synchronize files between their own devices, instead of using a cloud-storage solution. Within our use case, it performs well compared to the aforementioned commercial systems, but lacks the extra functionality and consistency that they provide.

Christian Thomas Hurley
<churle17@student.aau.dk>

Christoffer Hansen
<ch17@student.aau.dk>

Fredrik De Frène
<fdefre16@student.aau.dk>

Mads-Bo Bomholt Lassen
<mlasss17@student.aau.dk>

Mathias Andresen
<mandr17@student.aau.dk>

Simon Park Kærgaard
<skarga17@student.aau.dk>

Contents

1 Introduction

Personal data can be worth a lot of money to companies [1], and cloud storage services are potentially sitting on a latent gold mine. With personal data, companies can tailor advertisements around this data, increasing their effectiveness. Even if the cloud storage provider has no ill intent, both hacking and the CLOUD Act can be a potential threat to a person's data-integrity. The CLOUD Act allows federal law enforcement in the USA to demand US-based companies to turn over their data with a warrant or subpoena. This also applies to users not residing in the US but are using a cloud service based in the US [2].

Risk of data-loss is always present, as unexpected errors can occur to where the data is stored from numerous sources, e.g., mechanical failure or physical damage. Cloud storage services like Dropbox provide an easy but costly solution to mitigate data loss risk by ensuring the data is available on multiple devices and backed up at several geographical locations [3]. Cloud storage is increasingly popular and is estimated to be used by more than 2.3 billion people by 2020 [4].

The multiple data centers and internet infrastructure needed to support cloud storage services and their data loss precautions consumes excessive amounts of electricity. The United Nations calls for action by all countries to protect the planet, promoted through 17 sustainable development goals. To this end, we focus on the 12th goal that aims to ensure sustainable consumption and production patterns [5]. To illustrate the current trends and projections, we look into A. Andrae's findings on communication technology's global electricity usage [6]. He states the expected electricity consumption of data centers and internet infrastructure is projected to rise at a rapid pace until 2030 and beyond.

As depicted in Figure 1.1, data centers and wired networks are expected to account for most of the Internet's total electricity usage. The expected usage of 8000 TWh/year equals 20% of the world's expected total electricity usage by 2030. Andrae also alludes that if the correct measures are taken, the best-case scenario is that communication technology could only consume 8% of the world's total electricity consumption.

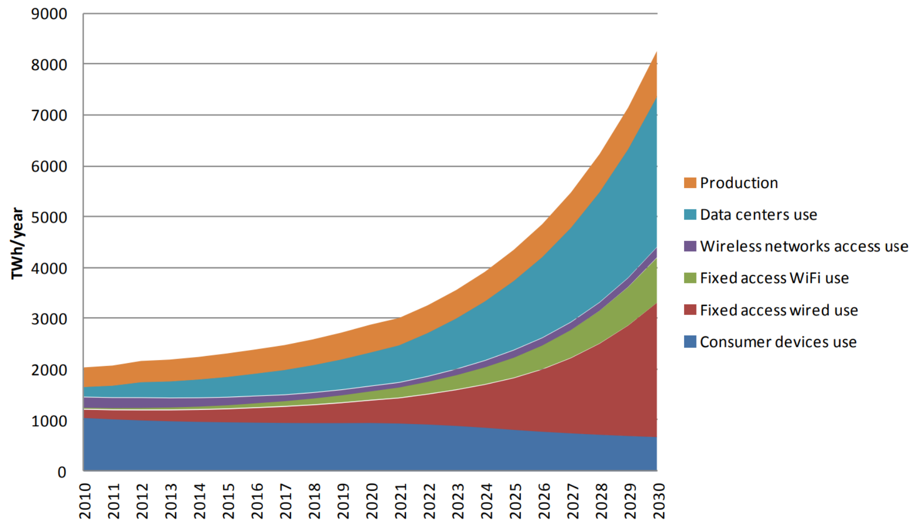


Figure 1.1: Expected communication technology electricity usage 2010-2030 [6, p. 140].

Figure 1.1 also shows that consumer devices' global electricity demand is expected to decrease in the future as they become more efficient. An alternative to investing in new, costly, and resource-demanding data centers to accommodate our increasing need for data storage, would be to use consumer devices to store and share data. This could also reduce global electricity usage and hardware production. Many consumer devices come with increasing amounts of storage that often exceeds what a user needs. A system using multiple users' devices for storage would, similar to DROPBOX, also have a data loss mitigation solution as consumers carry and have their devices located in multiple locations. This could be considered an unused resource that should be explored.

We explore the possibility of sharing information between private individuals without using commercial storage solutions. We pursue an alternative solution that uses existing consumer devices in order to accommodate the UN's 12th sustainable development goal. This leads us to the following initial problem:

Could there be an alternative to cloud storage services hosted by large companies, while retaining comparable benefits? Would it be possible to utilize the devices we already own and use, and achieve similar features and data safety?

2 | Problem Analysis

To familiarize ourselves with our initial problem, we investigate the aspects that define a file storage system for an ordinary consumer. To implement a solution, we must first understand the needs of a user and expect from such a system, and so we review a variety of predefined and customized user needs categories. In this context, we investigate existing classifications of file storage systems to deduce their strengths, weaknesses, and what challenges we might face when it comes to our solution.

2.1 User Needs

In order to build a beneficial file storage solution, one has to take the needs of the users into consideration. Based on the software quality characteristics presented by N. Bevan [7] and the requirements for distributed file systems defined in Distributed Systems: Concepts and Design by Coulouris et al. [8, Chapter 12.1.2], we define a number of user needs categories for a file system: *Usability*, *Reliability*, *Privacy*, and *User Transparency*. The categorization focuses solely on the user's perspective, not on the technical implementation or difficulties, and will be used to evaluate existing solutions.

2.1.1 Usability

Usability refers to how easy the user can operate the service. The term usability is based on the ISO definitions presented by Abran et al. [9, Table 1]. Based on the ISO definitions, we create our own definition, in which we have merged the functionality and efficiency characteristics by Bevan [7] into usability. From the user's perspective, increasing the system's functionality and reliability will improve the system's usability as it becomes more usable if it is more reliable. Likewise, the characteristics of maintainability and portability are included in usability, since a user does not care for how easy an error is to fix or how easy it is for the developer to make it run on another platform.

Following our definition, the possibility to install the service on multiple devices and simply log in to make the service functional, is an example of high usability. An example of lower usability is having to perform further setup, i.e. having to choose which devices should synchronize. The service being slow also affects the usability, as it will increase the time to synchronize files between devices.

2.1.2 Reliability

The definition of reliability is described as *"the capability of the software to maintain its level of performance when used under specified conditions."* [7]. We use this as a basis for our definition, upon which we extend with three characteristics from Coulouris et al.: *Concurrent file updates*, *Fault tolerance*, and *Consistency* [8]. When the reliability of a system is high, the system has high up-time, and works as intended. If two users are working on the same file at the same time, and one user's actions invalidates the work of the other user, this is an example of a *concurrent file update* and *consistency* problem. High reliability also implies that data loss should be prevented in the case of service or device failure, namely *fault tolerance* [8]. As a result of this definition, if the reliability of the system is optimal, the user should not worry about losing or overriding files and the files should always be accessible.

2.1.3 Privacy

Privacy covers the user's need to ensure that only those authorized by the user can access their files. This covers both third-parties such as hackers or data collectors, and first-parties such as the storage host itself. Privacy concerning third-parties is defined as security by Coulouris et al. [8], and we also name it security for the remainder of the report. For instance, if a man-in-the-middle attacker can intercept the data a user is sending to the service, this would be regarded as low security. Another example of low security is if the server on which a user's data is stored is hacked, and the infiltrator can access the data. A host scanning a user's photos for content to create better-targeted advertisements is an example of low first-party privacy.

While neither Coulouris et al. [8] nor N. Bevan [7] introduces privacy as a characteristic, we use it as an umbrella term to cover both security and first-party privacy. We focus on privacy since it is important for file systems, which is supported by the work of researchers at ETH Zurich and IIIT-Delhi [10]. They describe how users are generally aware of privacy risks concerning cloud storage. We see a reason for introducing privacy as a user need in Figure 2.1.

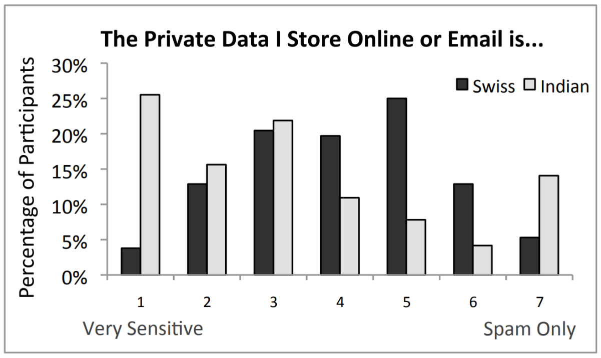


Figure 2.1: How many percentages of Swiss and Indian people are storing different levels of sensitive data in the cloud (Online or Email) [10, Figure 6]

It shows that less than 5% of the Swiss people who were questioned stored *Very Sensitive* data in the cloud. More of the Indian people who were questioned, answered that they stored *Very Sensitive* data (~25%). However, this could stem from political and cultural differences in Switzerland and India. In India, *"social and family structures place much less importance on privacy."* [10]. This study indicates a general need for privacy. People who already trust and use cloud services want privacy to ensure their data's integrity. On the other hand, people who refrain from using cloud storage do so because they do not trust the service to provide sufficient privacy.

2.1.4 User Transparency

When users can access information regarding how the service operates and where their data is stored, the level of *user transparency* of the service is high. It can also be that the service will inform its users in the case of a security breach, where the service cannot ensure that unauthorized parties have not accessed their data. An example of what can be perceived as reduced user transparency is how many large web services are using AWS (Amazon Web Services) for their servers, without informing their users. Suppose a person wants to discontinue their use of Amazon services. In that case, they would benefit from a companies being transparent regarding their partners. As an example, Netflix uses AWS for almost all its storage and computing [11], but it is not initially clear from their website. Concerning cloud storage, one of the prominent providers, Dropbox, hosted many of its services on AWS in 2012, according to a paper by Idilio Drago et al. [12]. User transparency does not encapsulate the technical *transparency* described by Coulouris et al. [8], which we classify as being part of usability.

2.2 Existing Systems

The following section provides an overview of different storage system key features, which we divide into three categories: centralized, cloud, and distributed. We do this in hopes of a more effective evaluation of each existing system within a category. In this evaluation, we use the four categories from Section 2.1 to compare the systems.

2.2.1 Centralized Storage Systems

A centralized system follows a client/server architecture, with the additional restriction of being centralized in one geographical location. This example can be represented by a simple file server that a number of clients are requesting data from. A request made to the server by any client module is received at the same physical location. After the request is handled, the response is sent back to the appropriate client [13, p. 41-43].

This architecture has a significantly greater potential for failure because of its centralization. A major concern is the client's ability to access the servers reliably. If the servers are experiencing issues regarding its ability to communicate with its clients, the service becomes unreachable, and clients have no way of accessing the data. This can happen in the form of a power outage, system failure, network failure, etc. Another significant concern is the availability of data. If the servers experience a hardware failure or any other form of unexpected data loss, the time it will take for the backed up data to be live again on the servers, is time the data is not available for the clients. Any of the issues mentioned above makes the system unreliable and vulnerable to downtime. Combined, the vulnerability is increased even further. The effects of a Denial of Service (DoS) attack is also magnified on a centralized system.

2.2.2 Cloud Storage Systems

To help categorize systems that otherwise will require information not readily available, we define cloud storage systems as a *perceived centralized storage system*, that from a user perspective, is centralized. We define it as such to reflect users' perception of the system. For instance, Google Drive might seem like a centralized storage system, where a user upload and download their data from the same server, i.e., their Google Drive page, across multiple clients. However, behind the scenes, Google might be distributing their data across multiple servers in different geographical locations.

Some of the largest providers of cloud storage systems are Google Drive, OneDrive, and Dropbox [14]. We refrain from drawing any further conclusions on these systems in regards to their infrastructure, as the software is proprietary with limited available documentation. However, we define these as cloud storage systems, as they all function in a similar fashion to the scenario described in the above paragraph.

2.2.3 Distributed File Systems

A key goal in distributed systems is to have shared resources. The most important resource to share is perhaps the stored information within the system [8, p. 522]. We have previously looked at centralized and cloud storage systems and how they distribute files from a locally stored server, or servers, to clients around the internet and how this approach poses challenges regarding load balancing, reliability, availability, and security.

When a single company controls the servers, it poses additional complications in regards to censorship, DDoS (Distributed Denial of Service), and other types of malicious attacks. A problem with censorship is that centralized or cloud servers are vulnerable to national governments restricting access to big companies like YouTube, Google, and Wikipedia. The latter case is illustrated by the Turkish government, when they recently banned Wikipedia nationwide. They have also been accused of blocking access to social media sites like Twitter and Facebook [15]. While these cases do not concern any of the big personal cloud storage providers like Google Drive, OneDrive, or Dropbox, they still prove that our online social profile or files stored in cloud storage, which we often deem personal and in our possession, are not directly administrated by ourselves i.e. they have low user transparency. Additionally, it poses the risk of letting the host exploit the user's private data for their own benefit.

Several distributed file systems aim to improve on these flaws and combat the problem of an entire service being unavailable from having its centralized source of storage denied access to its users or vice versa. The idea of a distributed storage system is to have multiple locations where the data is stored. With this approach, it becomes harder to lose access to all data, whether this is the intention of a national government, hacker, or part of the system failing. Even if access to one or more entities within the system is lost, it will not limit the data's overall access. The idea of distributed systems is not a novel concept. Academic experiments of such systems have been conducted in the early 2000s, but commercial peer-to-peer systems are yet to be widely applied relative to the aforementioned cloud storage systems. Each of the following paragraphs deals with case studies of distributed file systems that deploy different strategies to achieve a persistent and reliable file-sharing system.

OceanStore OCEANSTORE intends to provide a worldwide distributed file system for use in a variety of applications such as hosting, databases, and other applications involving the sharing and persistent storage of large numbers of data objects [8, p. 451]. It uses a peer-to-peer system that has the design goals of supporting untrusted infrastructure and nomadic data. An untrusted infrastructure denotes that it allows new devices to be added to the existing system. The nomadic data means that information can flow freely between the system's distributed components and is untied from its physical location [16]. OCEANSTORE uses a version-based archival storage system, which keeps track of each update to all data blocks in a read-only form. This is encoded and saved on multiple servers within the system. A fraction of the encoded updates to the data blocks is sufficient to reconstruct the entire object. This makes all the data stored within the system persistent. To encrypt the data and provide consensus across replicas, the Byzantine-agreement protocol is deployed [16, 17].

Syncthing SYNCTHING is a free open-source continuous file synchronization application. It allows users to synchronize files between two or more devices. Similar to OCEANSTORE, SYNCTHING uses a peer-to-peer approach, but the system is designed more towards individuals and personal storage and gives the user full control. That also includes a BYOD (Bring Your Own Device) policy, which means the user has to provide the hardware/devices that should run the system [18]. To synchronize the files between the devices in a cluster, SYNCTHING keeps a "local model" of all files on each device that together forms the "global model". To keep track of the local models with the highest change version to update the global model, SYNCTHING uses its own BEP (Block Exchange Protocol) [18].

InterPlanetary File System IPFS (INTERPLANETARY FILE SYSTEM) is a peer-to-peer distributed file system with the goal of distributing the entirety of the internet to enable persistent data availability. It aims to preserve the history of the internet, as the data is currently lost when the host decides to close its servers. To do this, IPFS distributes pieces of content to all nodes within the global network that they define. Using content-addressing, as opposed to location-addressing, it is possible to identify each file in the global namespace. When each file has a unique encoded hash, it is possible to receive information from any node in the network with the requested hash. This ultimately increases the rate of received information because the requested information is sent from the closest node holding that specific information [19]. Each file can be divided into smaller blocks of data and stored on separated nodes within the system. IPFS then utilises a directed-acyclic datastructure (Merkle DAG) to connect the blocks of data stored on nodes. This is useful since when having to update certain parts of a website or file, it is only necessary to change the specific blocks instead of changing the entire file [20].

2.2.4 Evaluation of Existing Systems

We looked at centralized, cloud, and distributed file storage systems and identified their strengths and weaknesses. From the four categories we identified in Section 2.1, we evaluate each of the existing systems' strengths and weaknesses. To encapsulate our findings, we refer to Table 2.1.

	Usability	**Reliability**	**Privacy/Security**		**User Transparency**
Centralized Storage System	High	Low	Low	Medium	High
Cloud Storage System	High	High	Low	High	Medium
OceanStore	?	?	High	High	Low
Syncthing	Low	Low	High	Medium	High
IPFS	Medium	Medium	High	High	Low

Table 2.1: Evaluating the existing system based on the criteria presented in Section 2.1.

We have graded the systems from low to high in each category to identify their strengths and weaknesses, thereby learning from each system design's side effects. In regards to OceanStore, we refrain from drawing any conclusions in some categories as the system is no longer in production.

Usability The centralized and cloud storage systems offer easy access and setup for the average user. They are well integrated with a strong infrastructure across the internet and support cross-platform use. These characteristics, which are important to the usability, are not yet well established in the investigated distributed systems. To operate such a system requires more actions from a user to make it functional and connect multiple devices. However, IPFS has developed a GUI for their application to support the user, and IPFS has their own website with an installer, while Syncthing distributes their software through GitHub. This is why IPFS scores higher than Syncthing.

Reliability Section 2.2.1 explains why centralized storage systems are graded low in this category, as they only have one point of failure. In Section 2.2.2, we look into how cloud storage systems are less susceptible to failures due to the redundancy of storing data at multiple geographical locations. Distributed systems come with another dilemma. The systems we looked at rely on having at least one device with a certain piece of information active at all times for it to be available. Both Syncthing and IPFS are susceptible to this problem, but IPFS distributes all information across a much larger network.

Privacy Centralized and cloud storage systems score a low level of privacy, as they have a third-party owner of the hardware that the data is stored on. Especially for cloud storage systems, the level of privacy the users have, depends on the integrity of the company that stores the data. Companies that maintain their own data centers also have access to all stored information which in theory could be exploited. This problem is solved by the approach of distributed systems, as explained in Section 2.2.3. This is why the three distributed systems score high in privacy since it is possible for the user to choose themselves which devices the data is stored on. In regards to security, the evaluation of each service varies. The benefit of data centers is that it allows for additional security measures to be taken. If a cloud storage system becomes compromised, it is likely to affect large parts of their user-base. The distributed systems on the other hand, have issues with security when the user is in charge of protecting their own data. All of the evaluated distributed systems take security measures in form of strong encryption technology.

User Transparency The centralized systems have a high grade of user transparency, as the user knows exactly which server their data is stored on. Big cloud storage providers do not let the user explicitly know what location or on which server their data is located, as it is stored in multiple locations. The user is only aware that the data is stored at the company they subscribe to, and even in cases like discussed in Section 2.1.4, the data can be stored at a third-party company. Similar to reliability, SYNCTHING and IPFS scores differently because of their implementation. SYNCTHING is graded high because this system allows the user to select the devices for data storage. With IPFS and OCEANSTORE, it becomes non-transparent to the user on which exact device the information is stored, similar to cloud storage systems.

s

2.3 Problem Statement

Throughout the problem analysis, we have investigated the user needs and evaluated the existing storage systems in relation to the identified user needs. During the introduction in Chapter 1 we exhibit how high the energy consumption of data centers and the internet are, and how it will only continue to rise due to its increasing size and with that, a growing demand of storage solutions. In response to this development, we propose the idea of creating a storage solution that only utilizes devices that are owned by the consumers. Thus, making an alternative that does not contribute the the growth of data centers and increases the user transparency. It is clear that existing storage systems are lacking in some categories of user needs, as defined in Section 2.1, which must be taken into consideration in development of the new system. This brings us to the following problem statement:

How can an individual's unused storage resources be utilized when synchronizing files on a distributed system in a more resource efficient manner compared to traditional cloud storage providers, while maintaining scalability, high usability, reliability, privacy, and user transparency?

3 | System Requirements

In this chapter, we create a definition of our system as a solution to the problem statement. We define our system's expected behavior from which we construct a suitable architecture and at the end state a number of requirements. This behavior is formulated with respect to the user needs that are defined in Section 2.1 and other suitable features from our evaluation of existing systems in Section 2.2.4.

3.1 Overview

The proposed distributed storage system aims to provide individuals with a synchronized storage solution. The system takes advantage of existing resources such as excess storage and already online devices. It works as a backup tool for files, both binary and text, to synchronize and share between users.

One idea to help reach the UN sustainable development goal, presented in Chapter 1, is to utilize online hardware with unused storage as a relay device to synchronize files between different devices. Such a device could be a phone, as it is typically powered on most of the time. If we assume an overhead when keeping a device turned on, no matter if it runs one or multiple operations, it is cheaper in terms of electricity to use an already powered-on device. Furthermore, by using hardware that is already in use we can potentially reduce another strain on the environment in terms of production and, consequentially, electronic waste.

Motivated by our evaluation of the existing systems, presented in Section 2.2.4, we deem a distributed system suitable for our problem. A distributed system's strengths are applicable when reflecting on the user needs. While their usability are generally lower than a centralized system or cloud based system, it is made up for by the higher privacy, security, and user transparency. We argue that our solution will share similar strengths. To mitigate the deficiency of usability, we create a user-friendly GUI, as in the case of IPFS, that scores higher in usability due to giving the user a more accessible interface and control flow. We create the notion of a relay device that is meant to increase the system's reliability by functioning as an intermediate step for the changes without storing the actual files themselves. The idea is that more online devices will help the system avoid a worst-case scenario where no devices are online to distribute the requested files to a device.

We arrive at the following design goals:

- It should be possible to synchronize devices on different networks so that a user can use their devices at work, at home, as well as other places.
- The system should be easy to install, and set up on multiple devices across multiple platforms and connecting them should be seamless to keep the usability high.
- The system should not expose the user to long wait times.

The idea of making a distinction between storage and relay devices is useful if an often powered-on device does not have sufficient space for the data the user wishes to synchronize. It is also beneficial if one user can use another trusted user's device as a relay device. This alleviates the issue of the user not having their own server for coordinating file synchronization, as more relay devices would increase the chance of one of the user's storage devices communicating with a relay device.

In Section 2.1.3 we investigated how many users are aware of the privacy risks associated with using an online storage solution. Combined with Section 2.1.4, where we identified the ability for users to know where their data is stored, it leads to the idea of the user hosting the data storage platform themselves. The system's behavior is depicted on Figure 3.1, to illustrate how it works on a high level of abstraction.

3.1.1 System Limitations

While all properties described above are desirable, we delimit the system scope and refrain from looking further into all of them. These limitations adheres to:

- The system will not have any encryption at any levels of implementation.
- All devices are located on the same LAN instead of being able to communicate through the internet.
- The possible relay and storage devices only include devices running the Windows 10 operating system.
- The files to be synchronized are text files only.

We introduce the concept of trust, which we define as allowing a device to store the user's files. Furthermore, we have certain assumptions that shape the development of the system.

- Trust is based on the notion of reflexive closure i.e., if A trusts B, then B also trusts A.
- Trust is based on the notion of transitive closure i.e., if A trusts B, and then B trusts C, therefore A also trusts C.
- All devices within the network have the required storage capacity to store all files.
- All devices within the system will eventually come online.

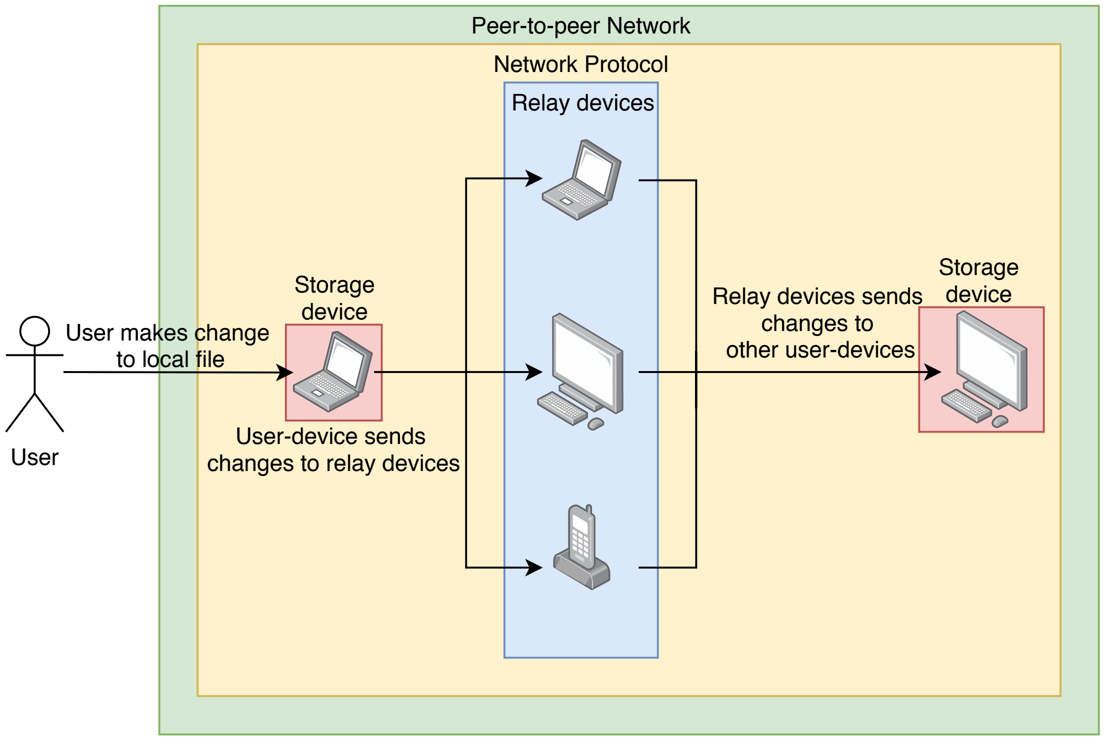

Peer-to-peer Network
Network Protocol
Relay devices
Storage device
User makes change to local file
User
User-device sends changes to relay devices
Relay devices sends changes to other user-devices
Storage device

Figure 3.1: An overview of the system. When a user makes changes to their files on their laptop, these changes are then forwarded to a group of relay devices. Each device within the system is annotated with either *Relay device* or *Storage device*, depending on their use-case. The laptop then sends the changes to one or multiple online relay devices, before the laptop is turned off. The relay devices hold these changes until the user turns on their desktop, that is also annotated as a storage device, in order to synchronize the files. In order to allow for these events to happen, all the devices, both relay and storage devices, must be in a trusted group with each other.

3.1.2 System Requirements

Based on the description of the system above, we create the following fundamental requirements, that the system should adhere to. The requirements are presented below and will be referred to throughout the design and implementation of the system.

R1 When any modification to a file or directory on a device is made, it should eventually be replicated on all of the trusted devices on the network.
This requirement encapsulates the most basic functionality of the system and is a core functionality of the system. A modification encapsulates all events that can happen to a file/directory such as it being created, deleted, and changed. It also expresses the user need of having a backup of their data, as discussed in Chapter 1. In the text above, we define what the system should be able to synchronize and share files between users and we reflect it here.

R2 **The replicated file version should eventually be the same across all devices.**
We expect the version of a file to be the same across all devices. Otherwise the user would risk losing data. It also requires the system to propagate the change in a timely manner, to not slow the user down when attempting to access the files.

R3 **Files are only synchronized and accessible on trusted devices that are specified by the user.**
This requirement reflects the user need of privacy and security. It is especially based on the arguments presented in Section 2.1.3, by allowing the users to keep their files on specific devices. It further augments the control and user transparency related to the user, by having them explicitly choose which devices their files are synchronized to.

R4 **The system allows devices to work as both storage and relay devices.**
As explained in the text above, relay devices will allow for the system to have a higher reliability. This requirement is also related to the UN sustainability goal with the idea of using already online devices as relay devices. As a relay device, the device will not be required to use the full amount of space all the files take, since it will only need to keep the changes made.

R5 **It should be easy for the user to interact with the system's functionalities.**
Similar to the existing distributed file systems, our system's level of usability will still be affected by having to configure multiple devices and connect them. However, part of this process can be streamlined by creating a user-friendly GUI for the user.

4 | System Overview

We design our system architecture and components based on the requirements defined in Section 3.1.2, ensuring that we achieve the user needs and intended system performance. In Figure 4.1, we illustrate an overview of the application running on each device while being connected to the network. The application is divided into two components: the *StorageComponent* and the *RelayComponent* as well as a number of shared services that is used by the two components. Within each component there exists a number of services too. We make this abstraction to limit the amount of internal communication between the individual services, that are not categorized as shared, and to minimize the dependency of services on other services.

The *StorageComponent* is responsible for keeping track of changes within the directory watched by the application as well as applying instructions from received delta files on the disk. When a change is detected on a file or directory, a delta file will be created and passed to the *RelayComponent*. The *RelayComponent* is then responsible for handling and distributing the newly created delta file to the other devices on the network. It is also responsible for receiving delta files from the network and pass them to the *StorageComponent* to be applied on the disk.

In the following chapters, we decrease the system design's abstraction level to look further into the behavior of the services that are running within the application's two components as well as the shared services. As a reference, the section number of each service can also be seen in Figure 4.1.

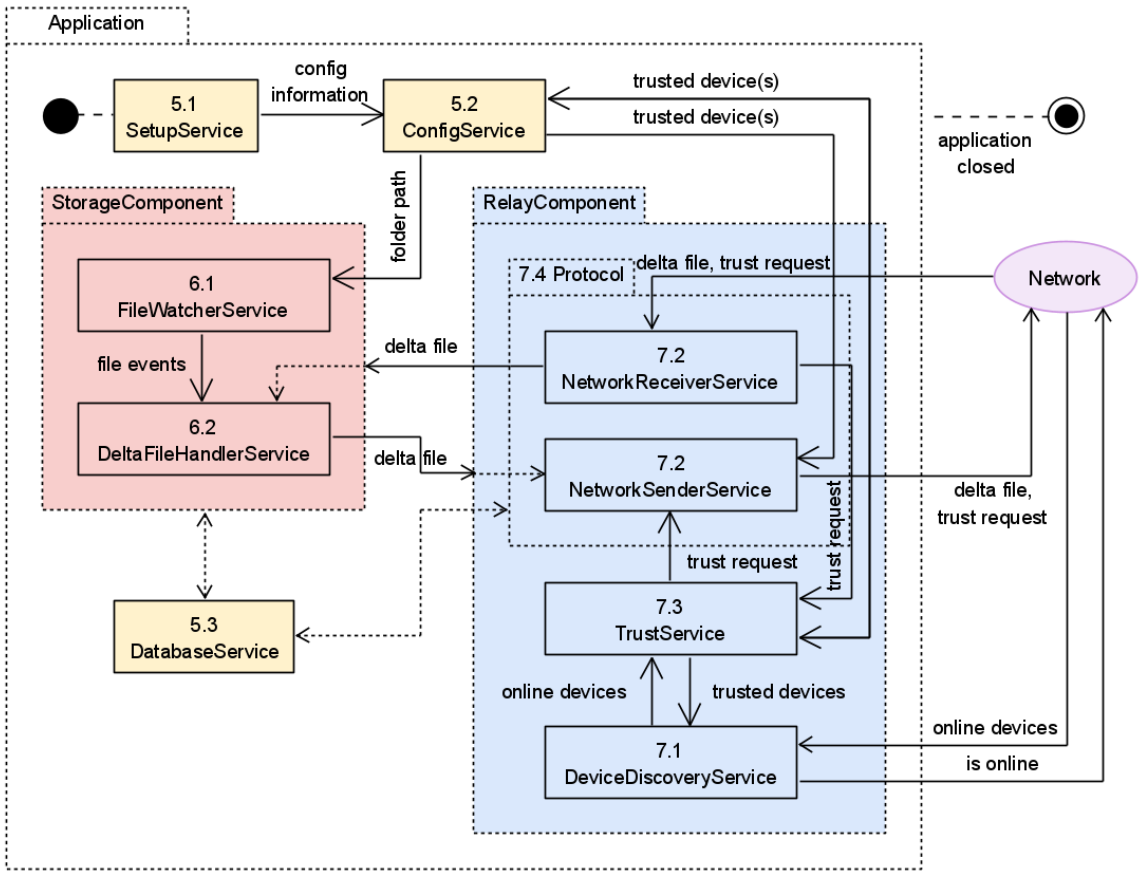


Figure 4.1: An overview of how data flows between services. The yellow services are shared between all components. The red services belong to the *StorageComponent* and the blue services are encapsulated by the *RelayComponent*.

5 Shared Services

The application's three shared services that are used by both the *RelayComponent* and the *StorageComponent*, namely the *SetupService*, *ConfigService*, and *DatabaseService* are in focus in this chapter. They are highlighted in Figure 5.1. These services are fundamental for the application and are thus the first services of our application, that we look at. We explain the design and implementation of each of them in this chapter.

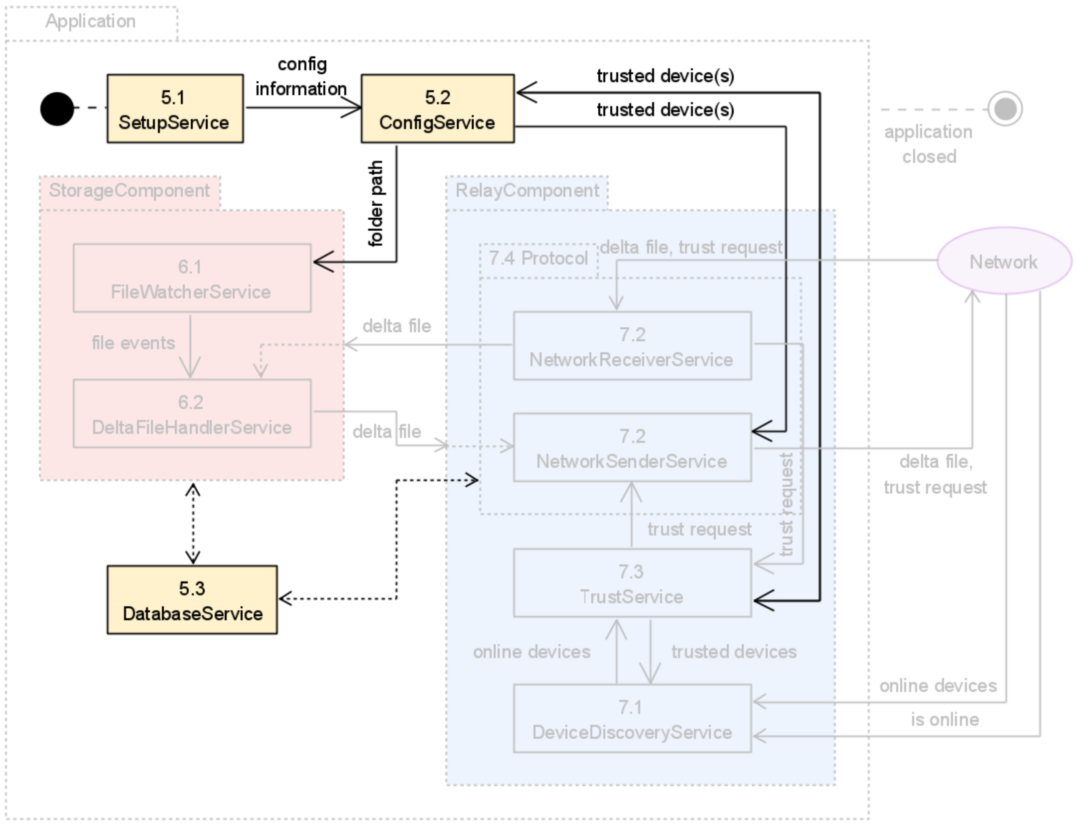


Figure 5.1: The shared services are highlighted to emphasize the focus of this chapter. The *ConfigService* and *DatabaseService* are closely connected with the *RelayComponent* and *Storage Component*.

5.1 Setup Service

The *SetupService* functions as the application's startup point. Its responsibility is to set up default values in the application via the *ConfigService*, as well as create the directories and files that the application uses. It initializes the *ConfigService* and creates the configuration file if it does not already exist. It iterates over the applications configuration settings to check if all required values are found and apply a default value to those that are not. After setting up the *ConfigService*, it checks if the folders used by the applications exist and creates them if they do not. For example, the directory used for synchronizing files should be created if it is the first time the application starts. The overall behavior of the *SetupService* is depicted in Figure 5.2

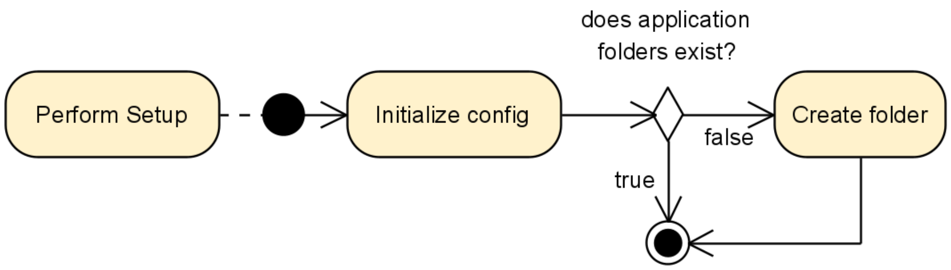


Figure 5.2: The *SetupService* runs every time the application launches to ensure correctness within the program.

In order to setup default values in the configuration at startup, the *SetupService* contains an array of tuples, where the first element is the key of the configuration, and the second element is the value, as we see in Code Snippet 5.1 in lines 1-4.

```
1  const defaultConfigs: [string, unknown][] = [
2    ["trustedDevices", []],
3    ...
4  ];
5
6  createConfig(): void {
7    defaultConfigs.forEach(([key, value]) => {
8      if (!configService.has(key)) {
9        configService.set(key, value);
10     }
11   });
12 }
```

Code Snippet 5.1: Default configurations in the *ConfigService*.

Then we go through each element in the `defaultConfigs` array and check if they exist in the *ConfigService*. If the value does not exist in the user configuration, we insert the default as shown in line 9. The reason for setting default values is to allow for new configuration entries to be added after the application has been running, for example, in an update. If we at some point decide to add a new setting, the *SetupService* will then make sure that a default value will be set before any service tries to access it.

After the *ConfigService* has been initialized, we go through the folders used by the application. This is the synchronization folder which defaults to `...<user>/dftp` and the application's AppData folder which is located at `...<user>/AppData/Local/dftp`. We check if the folders exist with the built-in FS module and create them if they do not.

5.2 Configuration Service

The CS (*ConfigService*) is responsible for writing to and reading from the persistent configuration. The configuration contains data that is affected by the user, and thus, the information handled by the CS will be different on each device. As we see in Figure 5.1, many other services depend on the CS since other services interact with the configuration. The CS is a simple CRUD[1] interface for key-value pairs that allow us to save data to storage based on keys so that the data can easily be retrieved again. The user can change application-specific settings such as the path of the folder to be watched by the *FileWatcherService*, and the name of the device. The CS will also keep the information used by the *TrustService*, such as the list of devices the user has specified as trusted. The service consists of four public methods:

- `set(key, value)` is used for setting the value of a key.
- `get(key, defaultValue)` is used for getting the value of a key.
- `delete(key)` is used to delete an entry from the configuration.
- `has(key)` is used to check if an entry is present in the configuration.

Instead of having separate create and update methods, we combine these into a single method called `set`, which will override the entry if one is already present. This makes it easy to set values, but means that to update an array, we have to get the array first, then mutate it, and then set the value again.

To save objects in persistent memory, we use the NPM[2] package CONFIGSTORE, which saves the data into a JSON file. The application's configuration file is saved in its AppData folder. This means that the user can change the values in the configuration manually, which could cause problems if the user does not know what they are doing. For example, if the user changes the trusted devices manually in the JSON file, it would invalidate the state of the program.

The problem can be solved by encoding the file in a non-human-readable format such as base64. Alternatively, we can also save the configuration in the database using the *DatabaseService*, but the

[1]Create, read, update and delete, the basic functions of persistent storage.
[2]NPM is the package manager for NODE.JS [21].

JSON format is convenient during development since it allows us to modify values manually and easily read them. Code Snippet 5.2 depicts the get method in the *ConfigService*, and how we fetch a value from the configuration.

```
1 public get<T = unknown>(key: string, default: T): T {
2     const value = this.store.get(key) as T;
3     return value || defaultValue || undefined;
4 }
5
6 const deviceID = configService.get<string>("deviceID", "defaultValue");
7 // or
8 const deviceID: string = configService.get("deviceID", "defaultValue");
```

Code Snippet 5.2: The `get` method and how to use it.

By using generic types in the `get` method, it returns a typed object, as shown in lines 1-4 in Code Snippet 5.2. Line 6 and 8 are equivalent since TypeScript can infer the generic type from the type of the assigned constant.

5.3 Database Service

The DBS (*DatabaseService*) is responsible for storing the specific version of the synchronized files on each individual storage device. It is designed in a way that allows us to change what DBMS[3] is used. Furthermore, it stores all the delta files that are sent to other devices on the network. The concept of delta files will be covered in Section 6.2. For each of the outgoing delta files, the DBS also stores the corresponding incoming acknowledgments, indicating that the delta file was received. During application startup, the DBS creates the tables necessary if they do not already exist, which we illustrate in Figure 5.3.

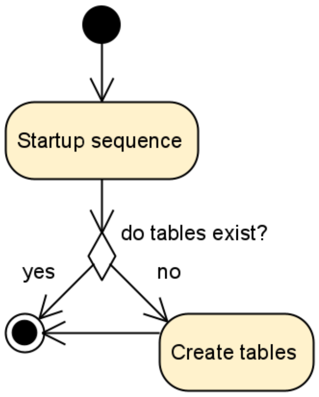


Figure 5.3: The *DatabaseService* will check if the database's required tables exist. If not, it will create them.

[3]Database Management System

TypeORM

We implement the database using a library called TypeORM. TypeORM is an Object/Relation Mapping tool that runs in Node.js [22]. The purpose of using TypeORM is to simplify interaction with the database. TypeORM works with multiple DBMSs such as MySQL, PostgreSQL, and SQLite [22]. We are using SQLite since it is small and serverless, meaning that it is not running as a separate process, unlike PostgreSQL. SQLite works by interacting with a single database file with a write lock on. In the application, the file is called `database.db` and is stored in the AppData folder for the application.

We initialize the database during the startup of the application. In order to create the schema of the database, we write our entities as TypeScript classes and annotate them with TypeORM metadata using decorators. As an example, Code Snippet 5.3 shows the definition of the `DeltaFile` entity.

```
@Entity()
export default class DeltaFile {
  @PrimaryGeneratedColumn()
  public id: number;

  @Column()
  public fileLocation: string;

  @Column()
   public version: number;
   ...
   @OneToMany(() => DeltaFileAck, ack => ack.deltaFile)
   public acknowledgements: DeltaFileAck[];
   ...
}
```

Code Snippet 5.3: Part of the `DeltaFile` class.

Defining the entities with TypeORM decorators means that they can be used to generate the database tables as well as objects in the application. This means that the schema of the database will always map to the classes used in the application. The `PrimaryGeneratedColumn` decorator denotes the primary key of the table and that it is auto-incremented. We use the `OneToMany` decorator to create relations between objects. The `DeltaFileAck` entity contains a `ManyToOne` decorator which is the counterpart to `OneToMany`. This means that in the database, the table for `DeltaFileAck` will contain a foreign key referencing the `id` field of the `DeltaFile` table. When the entities are set up, it is possible to interact with the database through simple queries using TypeORM's Repository API. As seen in Code Snippet 5.4, selecting a `DeltaFile` is possible with a `findOne` call to the `DeltaFile` Repository, which returns a `Promise`[4] for the first `DeltaFile` object that fits the criteria.

[4]A Promise object is an object that will eventually result in an object or be rejected with an error [23].

```
const deltaFile = await getRepository(DeltaFile).findOne({
  fileLocation: "<file>.txt",
  version: 1
});
```

Code Snippet 5.4: Selecting a `DeltaFile` from the database.

We insert entities using the `insert` operation on the repository, which takes an object of the entity class. With TypeORM, we can also write queries using a query builder and raw SQL queries. To encapsulate more advanced custom queries, we create a custom repository for the `DeltaFile` entity. The custom repository is a class that extends the `Repository` class from TypeORM. We can then get it with `getCustomRepository(DeltaFileRepository)`, where `DeltaFileRepository` is the name of the custom repository class.

The Schema

We define 3 entities in the application: `DeltaFile`, `DeltaFileAck`, and `StoredFile`, resulting in the schema generated by TypeORM shown in Figure 5.4. The diagram is annotated using Crows Foot Notation [24], where the relation between `delta_file` and `delta_file_ack` is depicted with a *1-to-optional-many* connection.

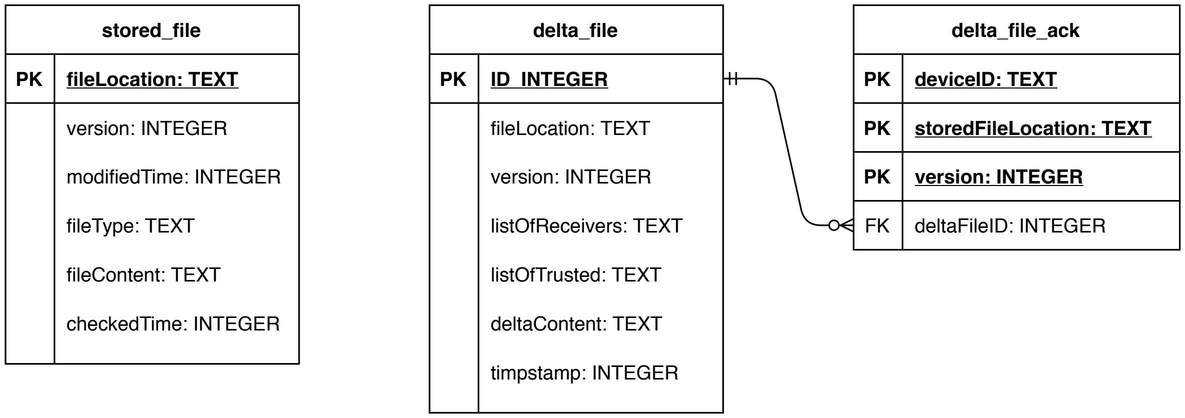


Figure 5.4: Database diagram of the local database stored on each user's personal device.

Each file and directory in the storage directory has an entry in the `stored_file` table with its `fileLocation` being the file's path relative to the storage directory, as a primary key. With this implementation, the primary key is the same in all the local databases in every trusted device that share this storage directory. The `delta_file` table also has a `fileLocation` column, but this cannot be a foreign key since a delta file can exist for a file that does not exist yet or has been deleted. For example, if a delta file is received with the instructions that a file should be created, there would be no entry in the `stored_file` table yet. Finally, the `delta_file_ack` table contains the acknowledgments, which are received from other devices. It uses `deviceID`, `storedFileLocation`, and `version` as a composite primary key. Besides that, it contains `deltaFileID`, which is a foreign key referencing the

`ID` attribute of the `delta_file` table. Alternatively, the `fileLocation` and `version` of `delta_file` can be used as a composite foreign key in the `delta_file_ack` table.

The reason for using a database instead of files is to abstract the data away from the user and increase application speed. By using a database, the user will not have two copies of the same file in their file system and will not be able to edit the wrong file by mistake. SQLite is supposedly 35% faster than the file system for reading and writing, according to the creators [25]. They also state that *"a single SQLite database holding 10-kilobyte blobs uses about 20% less disk space than storing the blobs in individual files." [25].*

6 Storage Component

The *Storage Component* is required for all storage devices and encapsulates the *FileWatcherService* and *DeltaFileHandlerService*, as highlighted in Figure 6.1. It provides the functionality of watching a specified storage directory and looks for changes made within its contents in order to construct delta files. Furthermore, it handles applying delta files as they are received from the *NetworkReceiverService* and updating the storage directory accordingly.

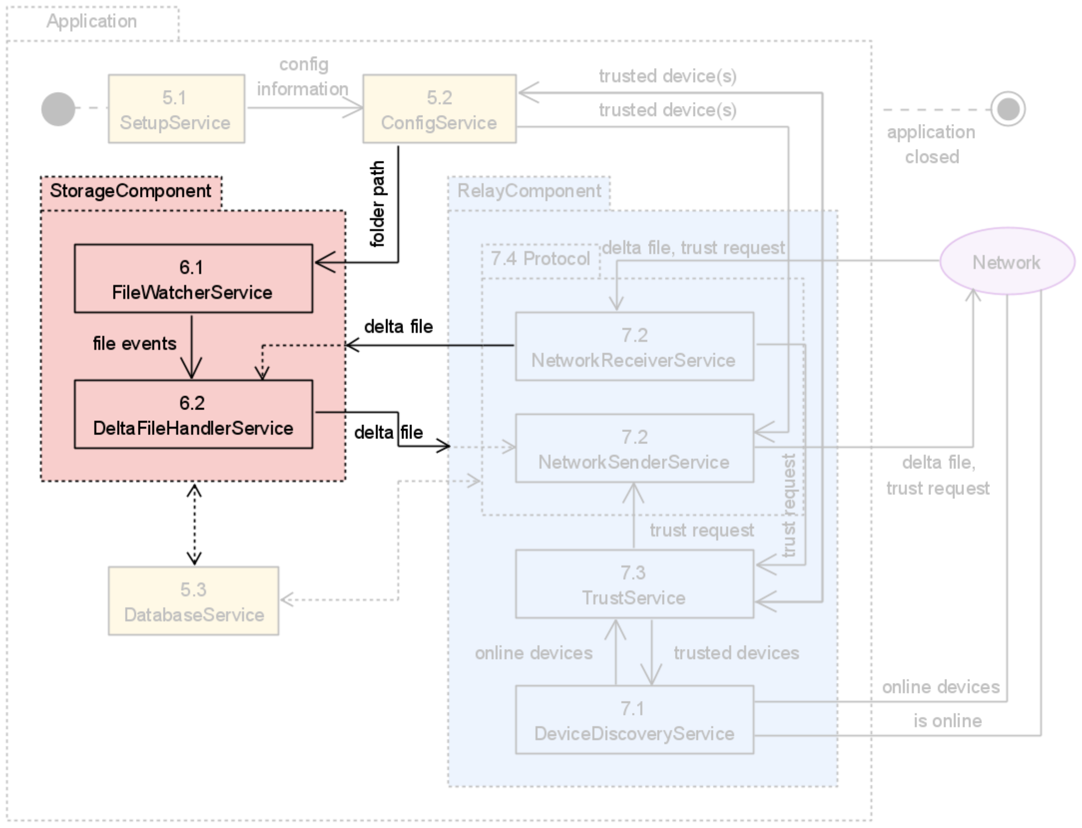


Figure 6.1: The *StorageComponent* is the focus of this chapter. It interacts with the *Relay Component*, *DatabaseService*, and the *ConfigService*.

6.1 The File Watcher Service

the FWS (*FileWatcherService*) is responsible for monitoring changes to the user selected storage directory. The FWS maintains a copy of each file, with metadata, within the *DatabaseService* to find differences when the storage directory changes. When these changes are detected, a corresponding delta file is created, and the FWS forwards this information to the *DeltaFileHandlerService*. In Figure 6.2, this interaction between the three services is depicted. In the same figure, the sequence, which we explain below, is labeled on the connecting arrows:

1. The FWS consults the *DatabaseService* for the state of the storage directory saved in the database, and metadata regarding each file. For each file 1..N, the database will have a corresponding copy $C.1..C.N$ including metadata.
2. Next, the FWS will give the specific event that occurred to the file, along with the file content of the changed file, to the *DeltaFileHandlerService*.
3. Lastly, the *DeltaFileHandlerService* finds the delta between the changed file from the previous step and the old copy it receives from the *DatabaseService* and inserts this into the *DatabaseService* again. We expanded upon this further in the following section.

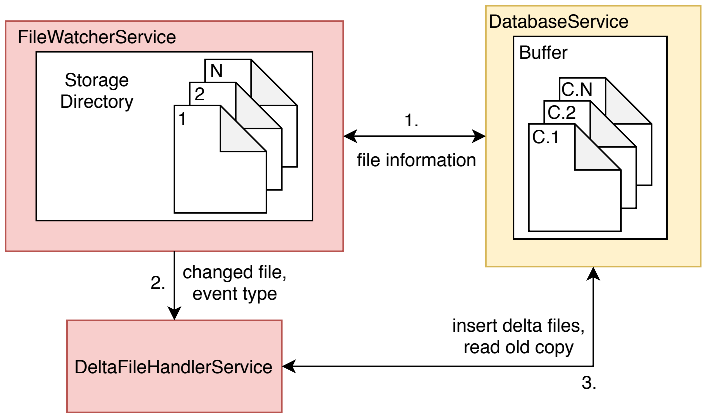


Figure 6.2: Illustration of the interaction between the *FileWatcherService*, *DeltaFileHandlerService* and *DatabaseService*.

The FWS continuously monitors the storage directory while the application is running. When the application is launched, the FWS inspects the storage directory to check if anything has changed since the application was running last, which implies the events of files being modified, deleted, and created.

All the files that appear in one device's storage directory appear in all device's storage directories that are part of the trusted group, under the same relative path name, and vice versa. Mirroring the folder and its contents come with certain challenges. Implementing the FWS requires us to observe all events occurring within the specified storage directory. We use the NPM package CHOKIDAR [26], an extension to the native NODE.JS file system interactions, to observe detailed events emitted by the file system. These events are presented and explained in Table 6.1. Our way of interpreting these events and relaying the information to the other devices makes the basis of our distributed file system.

Event	**Caused by**
add	A file is added.
change	A modification to the file is detected.
unlink	A file is removed.
addDir	A directory is added.
unlinkDir	A directory is removed.
ready	The *FileWatcher* finishes the initial pass over the directory.
error	An error occurs.

Table 6.1: The events that CHOKIDAR [26] outputs.

The FWS delegates tasks to the *DeltaFileHandlerService*, depending on which event is observed. It functions as a tool, which forwards information to the services that operate on this information. If changes occur when the application is not running, these changes are registered when the service turns on again. Additionally, since the FWS has an overview of the entire storage directory, it can provide new devices with all the files, directories, and their related information when added to the trust group.

Start-up Sequence

When the FWS starts as the application launches, the *ready*-event will initiate a recursive scan of the storage directory. In doing so, the FWS will check the modified time of each file and compare it to the `modifiedTime` stored in the database's `stored_file` table, seen in Figure 5.4 in Section 5.3. If the timestamps are matching, we update the *checkedTime* field. If they are not matching, we update the *checkedTime* field and send the information to the *DeltaFileHandlerService*, which will check if there is a difference between the modified file and the corresponding `stored_file` table entry.

When the directory scan is complete, the FWS checks for entries in the `stored_file` table that has not had their `checkedTime` updated, and therefore has not been visited. This means that the file has been deleted, and the corresponding entry in the database can be removed. This entails a delta file for that event is created and forwarded to the appropriate trusted devices to mirror the event.

Available Disc Space Requirements

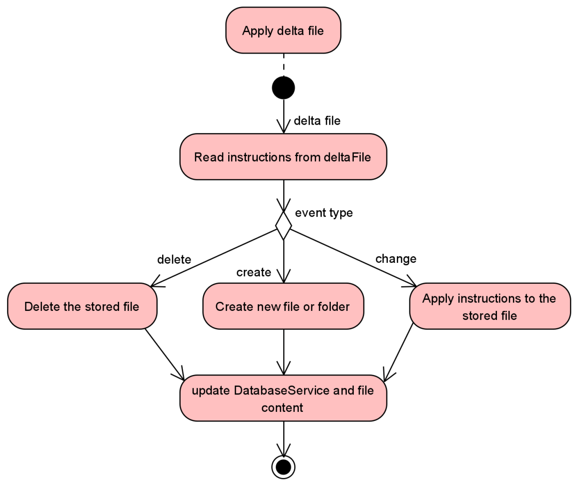
Whenever changes happen to files, we have to compare the new file to the old file in order to create the delta files. For this purpose, we save a copy of the content of each file in the database. This causes the synchronized files to effectively take up double storage space, which is a problem when the storage directory grows larger.

Alternatively, it would be ideal to intercept changes to the files in order to see the file before the change against the file after the change, instead of storing a copy. That allows us to create the delta file without using twice as much storage space. However, this solution will impose a new problem. If the file content is no longer stored, the application is no longer be able to detect changes that happens while the application is closed. That can possibly be solved by running a service at device start-up that runs in the background, but if that service crashes or is closed by the user, it invalidates the application's state.

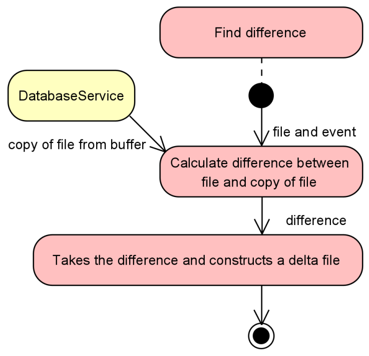
6.2 The Delta File Handler Service

The purpose of the DFHS (*DeltaFileHandlerService*) is to perform all interactions that regards delta files. This includes handling delta files from other devices on the network and creating new delta files within the application. The *FileWatcherService* provides the necessary information for the DFHS to calculate the difference between two files.

When we receive a delta file from the *RelayComponent*, the DFHS reads the instructions in the delta file and applies them to the corresponding file or directory in the storage directory. These two interactions are illustrated below in Figure 6.3a and Figure 6.3b.


Apply delta file
delta file
Read instructions from deltaFile
event type
delete
create
change
Delete the stored file
Create new file or folder
Apply instructions to the stored file
update DatabaseService and file content

(a) The *DeltaFileHandlerServices* receives a delta file and reads the instructions. Hereafter, it updates the file information in the database with the data in the received delta file.


Find difference
DatabaseService
copy of file from buffer
file and event
Calculate difference between file and copy of file
difference
Takes the difference and constructs a delta file

(b) Finds the difference between the provided file and the copy of the file from the *DatabaseService*, the service will then calculate and construct the delta file.

Figure 6.3: Diagrams showing the process of applying a delta file and finding the difference between two file versions.

Delta Definition

Now, we define what deltas and delta files imply and how it is beneficial to the DFHS. Delta encoding is the task of finding the differences between two bodies of data in order to transmit the least amount of information required. In order to fulfill our requirements **R1** and **R2**, which both revolve around synchronizing or replicating a file, delta encoding helps us create a structured representation of the changes, making them easier to apply. Delta encoding also helps us fulfill requirement **R4**, as relay devices do not need to store the full-size original files but just the content of delta files. Without using any form of compression, such as delta files, the concept of a relay device is not be possible since it would be equal to a storage device by storing the original files.

To correctly measure the difference between two files, we must first define what a file is and what it contains. We define a file as being equivalent to a string, where a string is a sequence of characters over an arbitrary alphabet. We will consider the content of all files to be strings going forward. The difference, or delta, of a string, can be found when the user edits the string. That edit turns the string from version s_{i-1} to s_i. The edit and delta can informally be described as:

$$\Delta = diff(s_i, s_{i-1}) \tag{1}$$

$$s_i = merge(s_{i-1}, \Delta) \tag{2}$$

The merge operation describes applying Δ to s_{i-1}, which results in the newest version of the string: s_i.

A multitude of algorithms exploit different strategies to solve this. These range from finding the longest common subsequence and shortest edit script to finding the shortest path in an edit graph [27]. Existing algorithms attempting to solve these problems are presented by E. Myers [27] and J. Hunt et al. [28]. Picking a solution depends on the desired behavior or run time of the service. Nevertheless, all algorithms considered should include the operations seen in Equation 1 and Equation 2.

Finding Delta

Adhering to Equation 1 and Equation 2, we utilize an implementation provided by K. Decker [29], which implements *An O(ND) Difference Algorithm and Its Variations*[27] presented by E. Myers in his paper on the same subject. K. Decker's implementation also provides other variations of the algorithm.

E. Myer's paper presents a solution to find, in an expected time and space of $O(ND)$, the shortest or longest path in an edit graph, where D is the size of the SES (Shortest Edit Script). This is further increased to a run-time of $O(N + D^2)$ under a stochastic model. However, it performs well on types of content that remains similar, like edits in a file. This algorithm finds the least amount of characters that should be changed and therefore has to consider many possible paths.

To find the shortest edit script, the algorithm traverses an edit graph of two strings being compared. An edit graph is depicted in Figure 6.4, where we find the shortest edit script to transform string *example* into *escape*. A valid sequence is one that always goes either $\rightarrow$ (Delete), $\downarrow$ (Insert), or $\searrow$ (Match) from $(0, -1)$ to (X, Y), where X is the length of *escape*, and Y is the length of *example*. The goal of the algorithm is to find the minimal traversal distance, which intuitively entices it to travel as diagonally as possible from the upper left to the lower right side. The other diagonal lines starting in the lower-left and ending upper-right show the size of the SES at each character in the sequence. In the case of Figure 6.4, the SES is the size of $D = 5$ at the end of the algorithm.

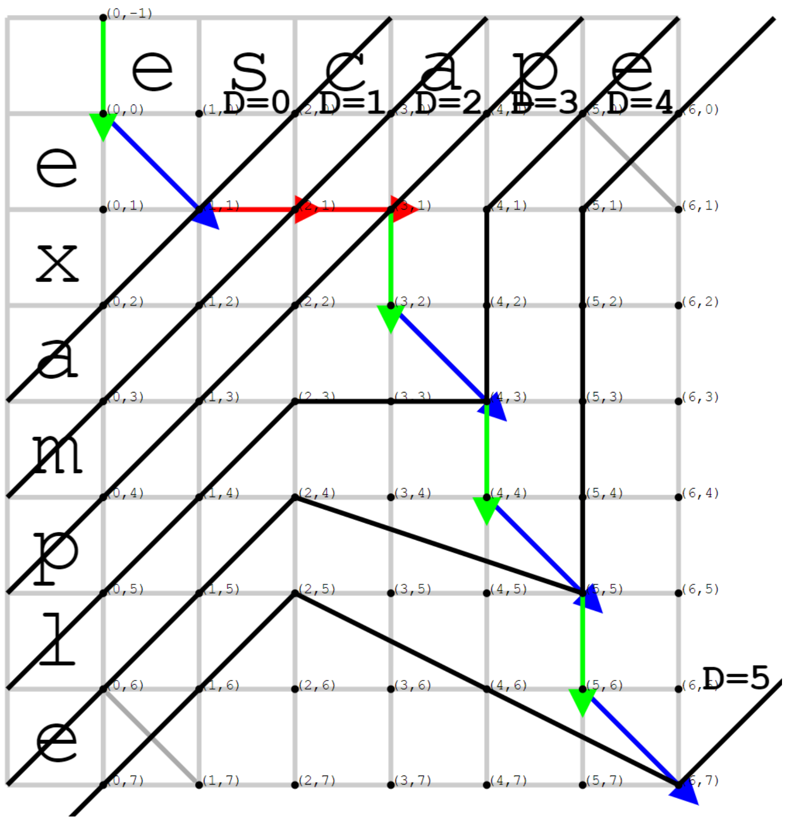


Figure 6.4: An edit graph of transforming $s_1 = escape$ into $s_2 = example$. The $\downarrow$ indicates a character is being inserted. The $\rightarrow$ indicates the character is deleted, and lastly $\searrow$ indicates the character is unchanged and is kept. The illustration is produced using an online tool [30].

Delta File Creation

In order to create a delta file, we define all changes as a set of instructions within the `DeltaCharacters` class, shown in Code Snippet 6.1.

```
export default class DeltaCharacters {
    character: string;
    position: number;
    isAdded: boolean;
    fileEvent: fileEvent;
}
```

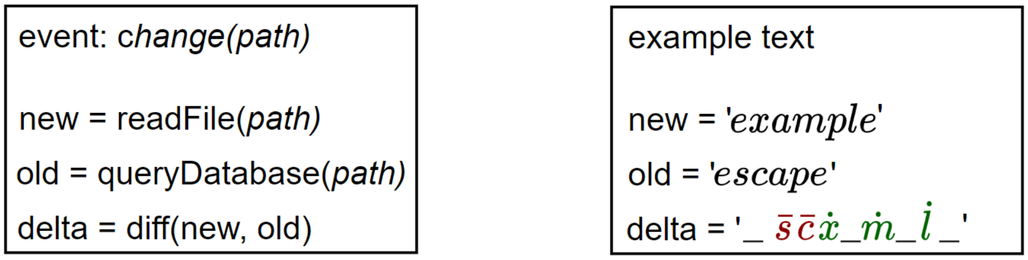
Code Snippet 6.1: The `DeltaCharacters` class contains information regarding what characters(s) are modified, their positioning, and whether they are added or removed.

The fields of the class represent the following:

- `character` denotes the character(s) of the change.
- `position` denotes the position in the file it is at.
- `isAdded` denotes whether the character(s) should be added or removed.
- `fileEvent` denotes that a file event can occur when the file is created or deleted.

Changes are read sequentially and stored in an array of `DeltaCharacters` objects. If the diff algorithm determines that a change consists of a sequence of characters, the `character` field can consist of multiple characters.

To illustrate how we find the delta between the file in the storage directory and the `fileContent` of the `stored_file` table, we present an example in Figure 6.5. When the *FileWatcherService* observes a *change* event, it forwards the event and path information to the *DeltaFileHandlerService*. The pseudo-code for the implementation is shown in Figure 6.5a with text example equivalents presented in Figure 6.5b.



```
event: change(path)

new = readFile(path)
old = queryDatabase(path)
delta = diff(new, old)
```

example text
new = '$example$'
old = '$escape$'
delta = '_ $\bar{s}\bar{c}\dot{x}$_$\dot{m}$_$\dot{l}$_'

(a) Pseudo-code example of a *change* event occurring on *path*, where we find the Δ of two strings.

(b) Here we present an example of a changing an old string into a new string based on the Δ between them. The Δ = 'scxml'.

Figure 6.5: A simple example of comparing two strings and the modification that has to be made to change the old string into the new one.

The *delta* in Figure 6.5b is the sequences of characters that have to be either removed, denoted by a line ($\bar{a}$), added, denoted by a dot ($\dot{a}$), or remain, which are the characters of *new* and *old* not contained in the *delta*. These are all tied to a `deltaCharacters` object's `character` string. We are only interested in sending the removed and added part of the *delta* and not the characters that remain the same. All information stored in the *delta_File* table is also put in the delta file, which is sent to the trusted devices via the *NetworkSenderService*.

This is the basis of how each delta files are constructed, but this is not a solution that takes minimal space. The `DeltaCharacters` implementation is not currently focused on creating small delta files. Instead, the current delta files have a larger representation that improves readability. This means that the delta file will sometimes be larger than the actual file, thereby defeating the delta file's purpose. For example, we see the DFHS converting `a-b-c` into `1-2-3` in Code Snippet 6.2. The use of newlines and indentation exaggerate the example, but even without whitespaces, the delta file contains 417 characters, while the actual file contains five characters.

```
{
    "fileLocation":"<path>",
    "version":1,
    "listOfReceivers":[],
    "listOfTrusted":[],
    "fileType":"file",
    "deltaContent":[
        {"character":"a","position":0,"isAdded":false},
        {"character":"1","position":0,"isAdded":true},
         {"character":"b","position":2,"isAdded":false},
         {"character":"2","position":2,"isAdded":true},
         {"character":"c","position":4,"isAdded":false},
         {"character":"3","position":4,"isAdded":true}
    ],
     "timestamp":1608764400000
}
```

Code Snippet 6.2: Delta file converting `a-b-c` into `1-2-3`

Nonetheless, this is not a problem for all cases. If we change a single character in a file that contains five paragraphs of text generated using `loremipsum.io`, we end up with the delta file seen in Code Snippet 6.3. Here the delta file contains 187 characters versus the 3907 characters in the actual file.

```json
{
  "fileLocation": "<path>",
  "version": 1,
  "listOfReceivers": [],
  "listOfTrusted": [],
  "fileType": "file",
  "deltaContent": [
    {"character": "p","position": 1856,"isAdded": true}
  ],
   "timestamp": 1608764400000
 }
```

Code Snippet 6.3: Delta file changing one character in a file with 5 paragraphs of lorem ipsum

Both delta files have a static overhead of 139 characters, not including whitespace, for everything not regarding the delta content. This grows more insignificant relative to the increasing number of changes, as the static information's size decreases relative to the content array. In the worst-case scenario, a delta file takes up $k + n \cdot c$, where k is the size of the static delta file overhead, n the number of characters, and c the size of the overhead on each delta content element. This happens if every other character is changed since that will create one added and one removed entry for every other character in the delta file, and no characters could be combined since there are no consecutive character changes.

Thus, a good way to make the delta files smaller would be to decrease c since that is included with every character changed. We can remove the labels, since that can be inferred from their positions, which will turn
`{"character": "p","position": 1856,"isAdded": true}` into
`{"p",1856,true}`. This already reduces the overhead from 42 down to 6, which is a ~85.71% decrease. We can also combine a removal and an addition at the same position into a single change. Besides decreasing the size of the file, we can also use a compression algorithm such as Huffman Coding to decrease the delta files' size during transfer. We choose not to focus on decreasing the size of the delta files since the overall concept of the delta files will be the same.

The current delta file implementation only supports text files and not binary files. This is because binary files are not readable and writable in the same way as text files are. In order to support binary files, we can encode them as Base64 and have that as the content of the file [31]. This will allow us to compare two versions of a binary file and send the data through our protocol.

Applying and Reversing Delta Files

When the *DeltaFileHandlerService* receives a delta file from the *NetworkReceiverService*, it must apply it to the corresponding file in the storage directory. The process of updating the file has the following steps:

1. Read the existing file in the storage directory with the delta file's *fileLocation*, if it exists.
    - If the file does not exist in the storage directory, then the file has to be created.
2. For each *DeltaCharacter*, mutate the string accordingly.
3. Send update to the *DatabaseService* requesting to update the *stored_file*'s file content.
4. Write the file in the storage directory.

In addition to these steps, there are checks for a *fileEvent* that dictates whether a file should be created or deleted, as well as matching the modified time of the newly written file and the database columns.

In order to revert changes to a file, we create a static method `reverse` on the `DeltaFile` class that can reverse a delta file. It takes a delta file as input and returns the reversed delta file. First we reverse the `deltaContent` array, and then flip the `isAdded` on each `DeltaCharacters` object. The reversed delta file can then be applied as normal.

7 | Relay Component

The *RelayComponent* is used for communicating with and transferring files to other devices. It must be present in all instances of the application, and it is designed to be able to work independently of the *StorageComponent* in order to enable us to more easily implement relay devices. The *RelayComponent* consists of four services: the *DeviceDiscoveryService*, *NetworkSenderService*, *NetworkReceiverService*, and the *TrustService*, all highlighted in Figure 7.1.

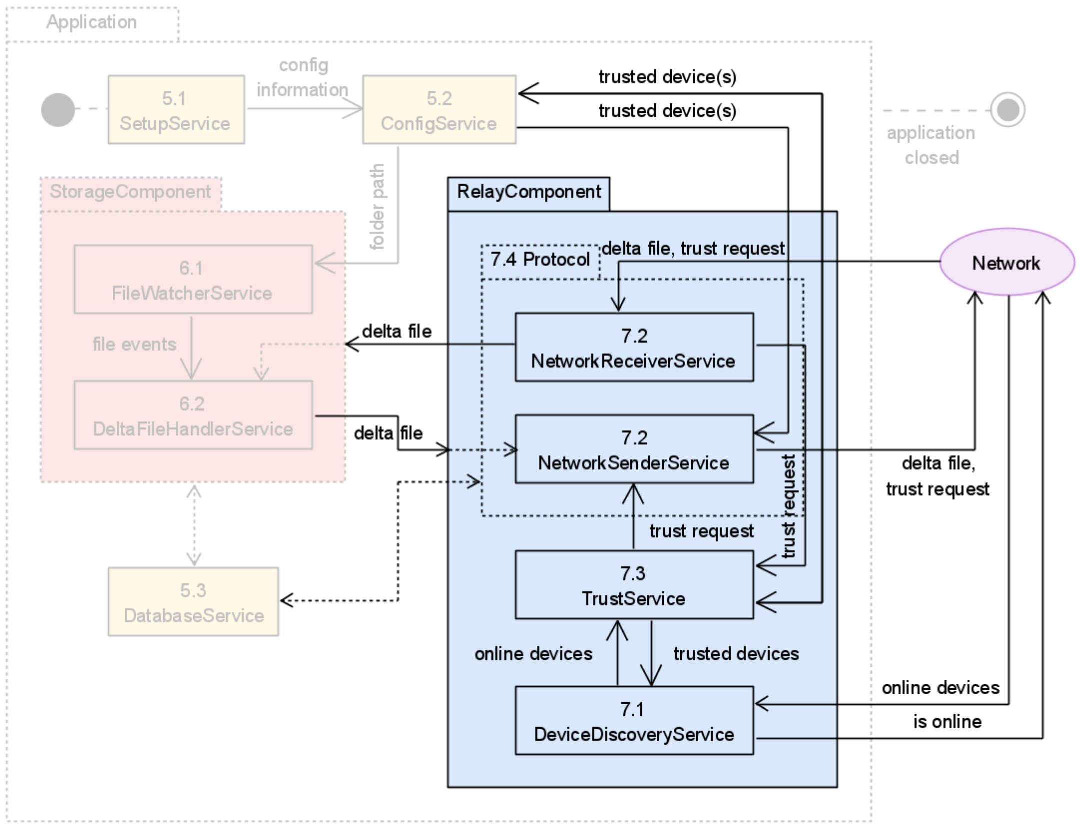


Figure 7.1: The *RelayComponent* is the connection between the network and the application's other component and shared services. Its underlying services and connections are highlighted.

7.1 Device Discovery Service

The DDS (*DeviceDiscoveryService*) has an apparent objective: Finding devices on the network that are running the application. The *NetworkSenderService* uses the DDS to contact a user's trusted devices and checks whether a trusted device is online and ready to receive messages. The behavior of the DDS is shown in Figure 7.2.

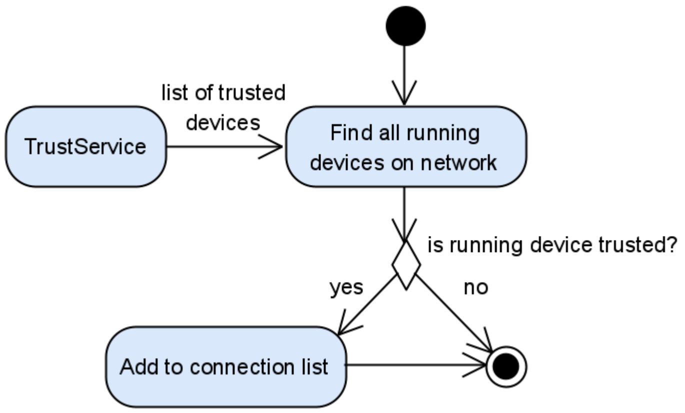


Figure 7.2: The *DeviceDiscoveryService* will find all online devices on the network and compare them to the list of trusted devices from the *ConfigService*.

The implementation of the device discovery process consists of two elements: The handshake endpoint and the *DeviceDiscoveryService*. We run a separate EXPRESS web server in the *NetworkReceiverService* that listens on a single `/running-check` endpoint. When another device sends a `GET` request to that endpoint, the DDS responds with a verification string, the id, name, and application version of the device running the endpoint. We use the verification string, which is a random UUID[1], to make sure that a device is running the correct application. This endpoint is used by the `pingDevice` function which is explained below. The *DeviceDiscoveryService* consists of three functions:

1. `pingDevice(device: Device): Promise<[boolean, Device]>` is used for testing if a device is running the application and returning the pinged device's information.
2. `findDevices(): Promise<Device[]>` is used for finding all devices running the application on the local network.
3. `findTrustedDevices(): Promise<Device[]>` is used for finding trusted devices that are running the application on the local network.

The `pingDevice` function uses the `fetch` function from the NPM package NODE-FETCH to send a `GET` request to the device's `/running-check` endpoint, mentioned above. The `pingDevice` function returns a tuple where the first element is a boolean telling whether or not the application is running. The second element is a `Device` object containing the pinged device's information. If it establishes a

[1]Universally Unique Identifier

connection, we compare the versions and check the verification string. In case of an application version or verification string mismatch, we return `false` as the tuple's first element. If there is no mismatch, we instead return `true` as the first element. In this case, we also assign a `Device` object containing the respondent's name and ID as the tuple's second element. If no connection is established, it returns `false` as the first element.

The `findDevices` function uses the `find` function from the NPM package LOCAL-DEVICES to find all local devices on the network. We wrap the `find` function in a web worker to avoid halting the application. The problem stems from JAVASCRIPT being single-threaded because it uses the event loop model for concurrency [32]. In most cases, this is fine, but it can create problems with demanding procedures. The LOCAL-DEVICES package uses the `arp -a` command to find a list of IP addresses, and then pings every address in the list to generate a list of online devices on the network [33]. We argue that it is this pinging that uses a lot of resources. Alternatively, we can run and parse the result of the `arp -a` command ourselves, and then ping the IP addresses using the `pingDevice` function. This might be able to decrease the load since we will not have to double ping anymore. However, as network optimizations are not the project's main focus, we choose not to study other solutions further since the current solution is adequate for demonstrating our system's functionality. Instead, to avoid halting the application, we wrap the `find` function in a web worker, which causes it to run in a background worker thread [34]. The worker then returns an array of `IDevice` objects containing the name, IP address, and MAC address of devices on the local network. Since the `IDevice` interface does not include an ID property, we create a `Device` interface extending the `IDevice` interface with the ID property. Then, `findDevices` uses the `pingDevice` function to check if each found device is running the application and returns all the devices that do.

We use `findTrustedDevices` to filter the list returned by `findDevices`, to find only trusted devices amongst all devices running the application on the network.

7.2 The Network Sender and Receiver Service

The NSS (*NetworkSenderService*) is responsible for emitting messages to other devices. The *DeltaFileHandlerService* uses the NSS when local changes are made, and the local device's trusted devices are sent a new delta file as a part of our Delta File Transfer Protocol. When a delta file is sent, the NSS uses the *DatabaseService* and the *ConfigService*, as shown in Figure 4.1, to fetch the corresponding message content and receivers. The NSS is also used by the *TrustService* to send trust requests to other devices. The list of trusted devices on each device dictates which devices the protocol is allowed to send delta files to.

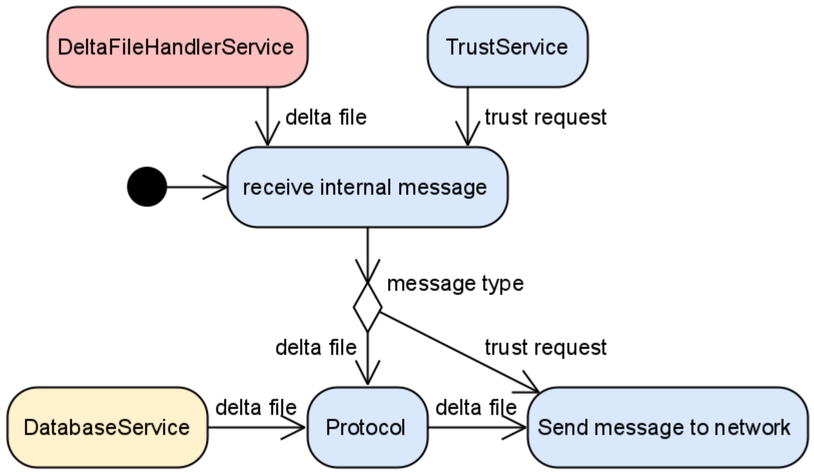


Figure 7.3: The *NetworkSenderService* receives messages from the application's two other services: *DeltaFileHandlerService* and *TrustService*. Delta files must be distributed by the *Protocol* before being sent to other devices on the network, while trust requests are sent directly to the receiving device.

The NRS (*NetworkReceiverService*) is the counterpart to the *NetworkSenderService*. It is responsible for handling all incoming messages from other devices on the network. As seen in Figure 7.4, upon receiving a *delta file* from another device, the NRS will send it to the *DatabaseService* and *DeltaFileHandlerService*, and when when receiving trust requests, it sends them to the *TrustService*.

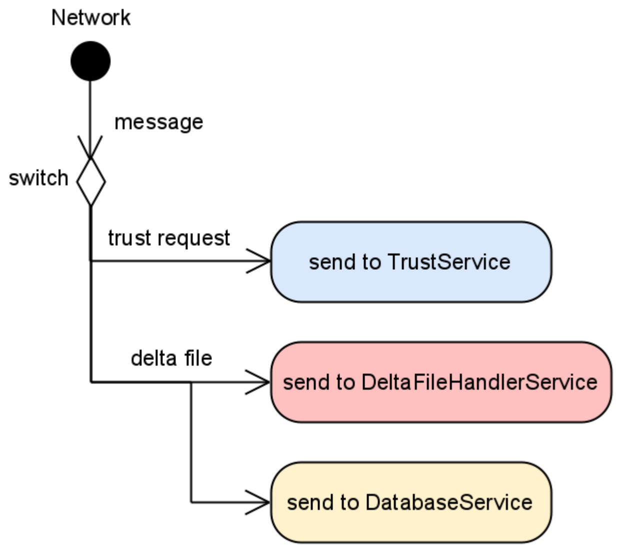


Figure 7.4: The *NetworkReceiverService* receives two types of messages from the network: Delta files or trust requests, and will pass them to the *DeltaFileHandlerService* or *TrustService*, respectively.

An integral part of the implementation of these services is the Socket.IO library. Socket.IO provides the opportunity for event-based real-time communication between devices [35]. Socket.IO is designed for client/server communication, so the application accommodates this by creating a web server. The web server makes the device running the application connectable and able to be communicated with using the SOCKET.IO library's client part. This is done through a `socket` object that can be used to emit events and oppositely listen for events. We use SOCKET.IO instead of the built-in UDP because of its expressed reliability [35], and its ease of use.

Since the *NetworkSenderService* is responsible for connecting to other devices, it uses the *DeviceDiscoveryService* to find online devices that it can connect to. In order to decrease the amount of device discovery calls that are made, we create a cached list of devices that we can connect to. We create a method called `findIPOfDevice` that takes the device we want to find the IP of and returns the IP if it finds it. First, we try to fetch the IP from the cached list. If it is not found, we use the IP of the device passed to the function. In case the latest known IP is defined, we ping it with the `pingDevice` function from the *DeviceDiscoveryService*. If we establish a connection, we save the IP in the cache list and return the result. If we do not establish a connection, or if the latest known IP is undefined, we call the `findTrustedDevices` from the *DeviceDiscoveryService*, which gives us the list of all online trusted devices. We then update the cache list with every online trusted device. Lastly, we return the IP of the device we searched for if found, and `undefined` if it was not found.

When we have the desired IP, we can instantiate the actual connection. Depicted in Code Snippet 7.1 in lines 1-12 is the `connect` method that creates a `socket` over HTTP. It uses the desired device's IP and adds the connecting device's id as a query to make it easier for the receiving device to identify the sender.

```
1  private connect(hostname, deviceID) {
2          const url = `http://${hostname}:${port}`;
3          const socket = SocketIOClient(url, {
4                  query: { id: this.id }
5          });
6          this.connectionMap.set(deviceID, socket);
7
8          socket.on("event", (args) => {
9              // Perform instructions related to this specific event
10              ...
11              socket.emit("response_event", ...);
12              ...
13          });
14          // More events
15 }
```

Code Snippet 7.1: The `connect` method and an example of a trust event.

When the `connect` method is called, the `socket` object it provides is used to emit events. The listeners for these events are set up when the `connect` method is called. In line 10 in Code Snippet 7.2, we emit an acknowledgment that the trust request has been received, which is an example of emitting events on the receiver side. Any event emitted by the *NetworkSenderService* will have an equivalent listener in this service that performs the desired operations for that event and vice versa.

```
private setupListeners(client: Socket) {
    const query = client.query;
    const id = query.id;

    client.on(event => {
            // React on event
    });
    client.on(EVENT_TRUST_REQUEST_SEND, device => {
          this.receiveTrustRequest(device);
           this.emit(EVENT_TRUST_REQUEST_RECEIVED, device);
     });
}
```

Code Snippet 7.2: The `setupListeners` method

As we defined in Section 3.1.1, we limit the system to only work between devices on the same network. If we are to make the system function over the internet, we will have to set up a proxy server or so-called handshake server. This would let devices see other devices running the application and create peer-to-peer connections.

There are two main reasons we choose not to do this. Firstly, using a proxy server will transgress some of the user-needs presented in Section 2.1, namely privacy and reliability, and user transparency to some degree, depending on how we inform the user about the server. Although it will only serve as a means of connecting devices, and no actual data will be transmitted through the server, we will still deteriorate the aforementioned user needs. Secondly, there will be scalability issues without sacrificing too much usability. When searching for devices to add to trusted devices, more instances of the application can mean an ever-increasing list of potential devices, essentially making it very difficult to find the right device to add at some point. A way to solve this can be to search for devices by supplying a unique ID or something similar. However, the usability of the application will be significantly reduced compared to selecting the desired device in a relatively small list. Granted, this can also happen if the application runs on a local network with many devices, such as if a large company or similar were to use the application. This scenario is uncommon on local networks but a constant factor on the internet, so we argue the application's current state retains a higher usability level than its hypothetical online counterpart. Alternatively, if there is a high demand for such a feature, the implementation is relatively straightforward because of the application's modular design.

7.3 Trust Service

The TS (*TrustService*) encapsulates the functionality of sending and receiving trust request by other devices, thereby establishing trust between two devices and their respective trusted devices. This adheres to our requirement **R3**. Hereto, we also refer to the two assumptions stating that our notion of trust is based on the reflexive and transitive closure. In Figure 7.5, we illustrate what happens when two devices, *Device A* and *Device B*, establishes trust, and how the *TrustService* handles it. In the example, each device is already a member of their own *trust group* consisting of 2..*N* devices.

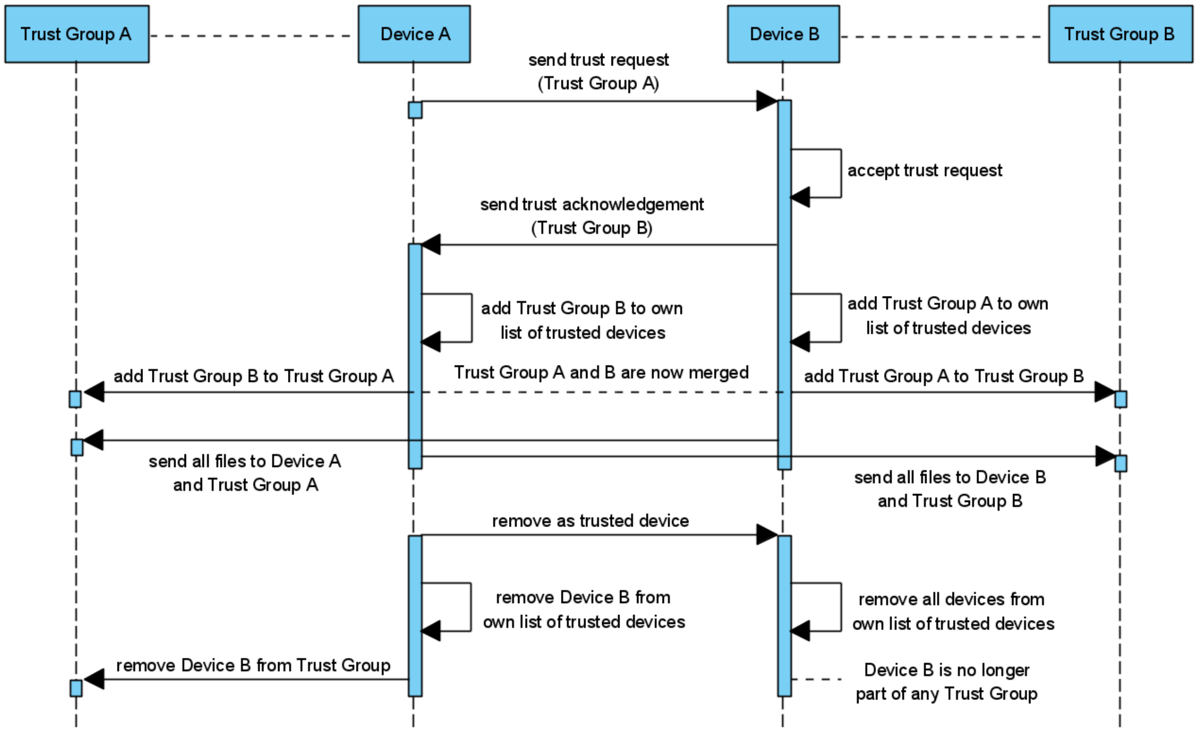

Trust Group A
Device A
Device B
Trust Group B
send trust request (Trust Group A)
accept trust request
send trust acknowledgement (Trust Group B)
add Trust Group B to own list of trusted devices
add Trust Group A to own list of trusted devices
add Trust Group B to Trust Group A
Trust Group A and B are now merged
add Trust Group A to Trust Group B
send all files to Device A and Trust Group A
send all files to Device B and Trust Group B
remove as trusted device
remove Device B from own list of trusted devices
remove all devices from own list of trusted devices
remove Device B from Trust Group
Device B is no longer part of any Trust Group

Figure 7.5: An UML sequence diagram of the trust process. Device A is initially a part of Trust Group A, and Device B is a part of Trust Group B. Upon both devices trusting each other, both trust groups are merged into a single trust group.

Device A initiates the process by sending a trust request to *Device B*, which accepts the trust request and replies with a trust acknowledgment. In each of these messages, they also send a list of devices contained within each of their respective trust groups to limit the number of messages being sent, and to disclose what other devices the device is accepting. Upon receiving a trust acknowledgment, both devices deems each other as trusted. Then *Device A* and *Device B* tells their trust group to trust the other device as well as the rest of its trust group, thus merging *Trust Group A* with *Trust Group B* to form a new combined trust group. At this point *Device A* and *Device B* initiates a file exchange and send their own files to the members of the opposite device's trust group. In the last part of the sequence, *Device A* removes *Device B* as a trusted device, which means the two devices no longer deem each other as trusted. *Device A* then tells all members of the combined trust group to remove *Device B* as a trusted device as well. Alternatively, *Device B* can also reject the incoming trust request from *Device A*, and end the sequence.

The implementation of establishing trust between devices and their groups is not encapsulated within a service of its own, as intended in our design. Instead, the *NetworkReceiverService*, *NetworkSender-Service*, *ConfigService*, and the GUI manages the trust process's logic. The *NetworkReceiverService* and *NetworkSenderService* are in charge of communication between devices when sending, receiving, accepting, or rejecting trust requests, while the user chooses the receiver of the requests from the GUI. The *NetworkSenderService* has the functions `sendTrustRequest`, `sendTrustAcknowledgement`, `sendTrustRejection`, and `sendDeviceToTrustGroup`. The *NetworkReceiverService* implements the counterparts to these functions. We can deduce the function's core functionality from the design; however, we implement further logic to mitigate unwanted side effects or trust failures.

Our design exhibits a sequential example where each device awaits the other to respond to a request or acknowledgment. We cannot guarantee this behavior in a live environment, as the user, through the GUI, handles the trust events from one device. When multiple devices can interact with each other on the network, the sequence of trust actions can be arbitrary. On each device, the user is able to send trust requests and remove already trusted devices from the GUI while another user might do the same. To mitigate problems that might arise in some scenarios of the trust process sequence, we keep track of three lists on every device: `trustedDevices`, `incomingTrustRequests`, and `outgoingTrustRequests`. These lists are stored in the configuration file on each device, as shown in Code Snippet 7.3.

```
{
    ...
    "trustedDevices": [
        {
            "ip": "192.168.1.1",
            "name": "DeviceName",
            "mac": "mac-address",
            "id": "unique-device-id",
            "trustedGroup": [...]
        },
        ...
    ],
    "incomingTrustRequests": [...],
    "outgoingTrustRequests": [...],
    ...
}
```

Code Snippet 7.3: The trusted devices part of the `config.json` file.

They aid us in creating solutions to cases where the scenario differs from the one depicted in Figure 7.5. Such a scenario could be that *Device A* sends a trust request to *Device B*, while *Device A* already has an incoming trust request from *Device B*. In that case, *Device A* will immediately send a trust acknowledgment to *Device B* and commence the steps of trusting each other, i.e., merging trust groups and exchange files. This check is handled by the `sendTrustRequest` function within the *NetworkSenderService*. Other cases, such as if a user tries to send a trust request to an already trusted device or a device that has already received the user's trust request, are made unattainable to the user through functionality in the GUI. Specifically, we remove the devices contained in the `trustedDevices` and `outgoingTrustRequests` lists from the list of devices that the user can send a trust request to from the GUI. Despite these measures, we still foresee problems with offline devices at the receiving end of a trust request, acknowledgment, or trust removal message. If both devices are not online simultaneously and one device proceeds to change its trust group, it will clash with the other device's trust group. Our solution to this is to ensure that the other device is online before sending a trust request to it or accept one from it. In short, the essential part of the trust functionalities implementation lies in updating the three aforementioned lists, `trustedDevices`, `incomingTrustRequests`, and `outgoingTrustRequests`, depending on what the user chooses to do.

The design choice of attaching *Device A's* trust group to the trust request it sends to *Device B* is made to increase the user transparency, as it allows *Device B* to see which other devices it accepts. An inherent problem with this approach is if *Device A's* trusted group changes before *Device B* accepts the trust request, then **Device B** accepts an older version of *Trust Group A*. This causes inconsistent trust groups, thus creating errors when synchronizing files. Our approach of connecting devices and their trust groups also has other side effects. Combining two devices' trust groups means that a single user can decide who has access to all the shared files on behalf of the entire trust group. We also force the trust group members to receive a number of files from the other device and its trust group. Our trust sequence diagram in Figure 7.5 also exhibits how *Device B* is left without any trusted devices after being removed by *Device A* despite it being a part of *Trust Group B* initially. These side effects are still reasonable under our system limitations, described in Section 3.1.1, where we assume that users trust each other and will be able to administrate the system's behavior accordingly. While a user may not want to let other devices access their stored files without their own explicit consent, we can not enforce this security. A user can always copy the files from its storage folder and share them with others.

Our procedure of requiring both devices to be online before any trust events can happen is a trade-off made during development. This trade-off adheres to a well known theorem used for distributed systems called the PAC-theorem. The decrease in *Availability* lowers our application's usability, but the increase in *Consistency* makes any trust-related event in the system much safer. There are many ways to come up with different solutions to let our users decide whom to share their files with and whom to trust. We present a few of the alternatives below:

- The closest alternative to our present solution is to let all members of the trust groups of the two devices willing to trust each other accept the opposite devices before a group merge and file sharing can commence. With this procedure, we allow all affected devices and their users to give their consent before sending their synchronized files to the other group as well as receiving theirs. This, however, is a tangled solution that will require a lot of communication between all devices and will increase the complexity of the trust procedure, which will escalate in relation to the size of the two trust groups.
- An arguable simpler alternative is to let our users create predefined and static trust groups. It requires the user to form a new trust group from the existing one if they want to expand the group. All the existing members then have to accept being a part of the new group. The same principle asserts oneself if a device has to be removed from a group. With this approach, we give each user the implicit choice of whom to share files with. To make this solution viable, it would require devices to maintain a separate storage directory for each trust group. This approach allows for a device to be a member of multiple trust groups simultaneously to serve different purposes, hence we give the users more flexibility. We would also conceivably not experience a device being left with no connections, provided it is a member of another group.

- Similarly, we can also create a new group whenever two devices have trusted each other. A third device may then request to become a part of that group, whereas all the group members will have to accept or reject it.

To sum up, we are not able to define a one-size-fits-all solution for our trust procedure. Numerous constrains could be further implemented to the trust process in order to achieve higher consistency, however these will come at the cost of the application's usability.

7.4 The Protocol

The protocol is responsible for making the device adhere to the policies of how delta files are sent and received by devices in order to reach consensus. The following sections provide the necessary theory that substantiates our policies, protocol design, and implementation of said design.

7.4.1 Multicast Communication

Multicast communication is an operation where a process in a group of processes sends a message to all other processes in that group. It is often used in distributed services. Requirements **R1**, **R2**, and **R4** from Section 3.1.2 can be solved partially through the use of multicast. In particular our solution to **R1**, benefits from multicast since it is useful for propagating changes to many devices.

According to Coulouris et al. [8, pp.169-170], multicast messages can help ensure the following four characteristics in a distributed system:

1. Fault tolerance through replication: A request is multicast to all (replicated) processes in the group, which again multicast the received messages in a similar fashion. This way, even if some processes fail, the request can still be serviced.
2. Discovering services in spontaneous networking: Multicast messages can be used to determine availability of services in the network.
3. Better performance through replication: If data is replicated to increase performance, multicast messages can broadcast new values as data change.
4. Propagation of event notifications: Multicast can be used to notify processes in a group about an event.

In our case, all four characteristics are useful to our system, and therefore we use multicast for a communication model.

7.4.2 Reliable Multicast

Reliable multicast builds upon the basic multicast primitives, namely *B-multicast* and *B-deliver*. In this context, delivered means a message is handed to the process's application layer, whereas received means it has been received but not yet delivered to the application layer. *B-multicast* will multicast a message to the rest of the group, and *B-deliver* will deliver a received message to the process's application layer. Basic multicast is not reliable since messages are immediately delivered as they are received. If the multicaster crashes, some processes might have already delivered the message to their own application layer, while other processes have not received it.

The primary way reliable multicast differs from basic multicast is that if a correct[2] process in a multicast group delivers a message, all of the group's correct processes will eventually deliver the same message. Reliable multicast has no default implementation, as its functionality can vary depending on its use. According to a paper on a reliable multicast framework: *"One cannot make a single reliable multicast delivery scheme that optimally meets the functionality, scalability, and efficiency requirements of all applications." [36].* The same paper proposes a minimal definition of reliable multicast to be *"the delivery of all data to all the group members, without enforcing any particular delivery order." [36].*

For an algorithm to be reliable multicast, it must satisfy three properties [8, pp.647-648]:

M.1 Integrity: A correct process delivers a message at most once, and the process is in the group that the message is intended for.

M.2 Validity: If a correct process multicasts a message, it is eventually delivered.

M.3 Agreement: If a correct process delivers a message, then all other correct processes in the group will eventually deliver the same message.

M.2 and M.3 combined ensure a liveness requirement since if a correct process eventually delivers a message, it will eventually be delivered by all the group's correct members. M.3 is also where the main difference between basic and reliable multicast lies. Basic multicast will deliver a message as soon as it is received, whereas reliable multicast will only deliver a message if all other correct processes eventually delivers the message [8, p.648].

7.4.3 The Delta File Transfer Protocol

To make sure that all devices in a network can reach consensus on files and thus synchronize the files correctly, we design a lightweight version of a reliable multicast protocol and expand it to the application's needs. We relax M.3 as the message is delivered before multicasting it. In our protocol, we define a multicast message, as being a delta file. A message is delivered once the delta file is applied. When a delta file is created, it is because a file was changed and therefore the message has already been delivered to the author of the message. If a process is to crash during the time between

[2]A process that has not crashed [8, p.650].

the delivery and the multicast, M.3 would not be met. This is, however, accounted for to some extent with the delta file exchange event that will be explained later.

Since users can make changes to their files while other devices are not connected to the network, there needs to be a mechanism for devices coming online again to catch up and reach consensus with the other devices. When we describe a device as being online or offline, we refer to if the device's application is either running or not running. It does not adhere to having a connection to the network. This is also needed if a user makes changes to files while not connected to the network itself.

As we explain above, it is necessary to relax the *Agreement* property. When implementing this for our system, we look at the reliable multicast algorithm from Coulouris et al. [8, p. 648] and expand it to conform to our needs. The following sections explain the different scenarios that can happen in the application and how it is implemented. For our delta file protocol, we create six events which can be seen in Table 7.1.

Event	**Description**	**Arguments**
offer_deltafile	Offer delta file to device	file path, version, hash
accept_deltafile	Accept a delta file offer	file path, version
send_deltafile	Send a delta file	delta file json
ack	Acknowledge that a delta file has been received	file path, version, deviceID
offer_exchange	Offer exchange	array of {file path, version, hash, isDir}
conflict	Initiate conflict handling	file path, version, received delta file json

Table 7.1: The possible events that can be emitted and received in the Delta File Transfer Protocol

Sending a Delta File

The flow of sending a delta file is shown in Figure 7.6. When a new delta file is created and ready to be sent to the devices on the network, an *offer_deltafile* event is sent to them, containing the file path, version, and hash of the offered delta file. To create hashes, we use the built-in library CRYPTO, which generates an MD5[3] hash based on the content, file location, and delta file version. The sender, which in Figure 7.6 is *Device A* reaches out and sends an offer for the oldest delta file that the receiver has not acknowledged yet.

Initially *Device B* acquires an internal lock identified by the file path, that the offered delta file adheres to. The lock is acquired to make sure that only one delta file is handled for each stored file at a time. In case we cannot acquire the lock, the request is put in a queue, which is served once the lock is yielded. When the lock is acquired, we query for the stored file at the file location. If either the stored file does not exist or the delta file version is greater than the stored file's version, meaning that the delta file has not been handled yet, we query the database for the delta file with the highest version. If no delta file was found, we emit an *accept_deltafile* event to *Device A*. If a delta file is found, and their versions are not equal, we also emit an *accept_deltafile* event to *Device A*. If their version are equal, we check if their hashes are the same, if so, we emit an *ack* event to *Device A*. Otherwise, we

[3]MD5 is a 128 bit hash function [37]

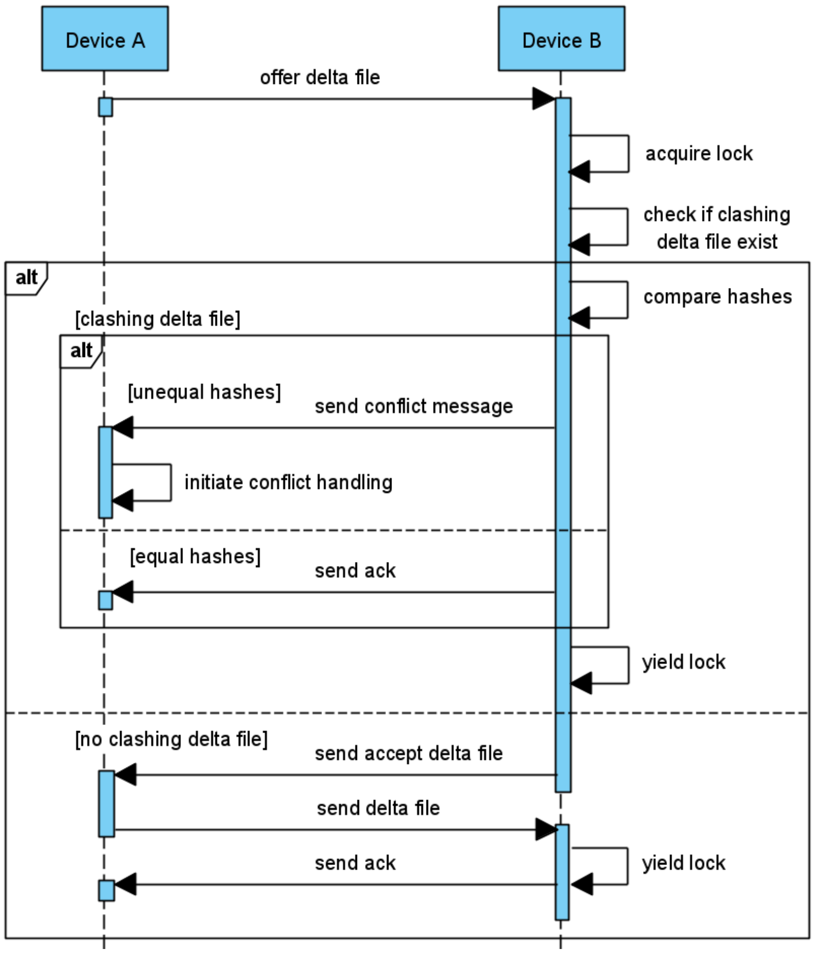


Figure 7.6: An UML sequence diagram showing the flow of sending a delta file between 2 devices.

have encountered a conflict, and will emit a *conflict* event to *Device A*, which then initiates its conflict handling, which will be explained later. The lock will be yielded after checking the hashes.

When the *accept_deltafile* event is received on *Device A*, the database is queried for the delta file, which is then JSON serialized and sent back to *Device B* with a *send_deltafile* event. When the *send_deltafile* event is received, *Device B* will deserialize the JSON into a `DeltaFile` object. Then it will query the database for the stored file matching the delta file. If there is not a stored file or if the delta file is the successor to the previous one, the delta file will be acknowledged and delivered by sending it to the *StorageComponent*, else an error has happened that is logged. We save an acknowledged from *Device A* in *Device B*'s database since *Device A* sent the delta file to *Device B*. *Device B* then checks whether it has received acknowledgments from all trusted devices, and deletes the delta files if it has. When receiving the acknowledgment on *Device A*, it inserts that information into the database, and makes the same check as described above. After sending the acknowledge *Device B* offers the delta file to all devices on the network that has not yet acknowledged it, from *Device B*'s point of view. Finally, the lock that was acquired during the *offer_deltafile* handling is yielded.

The system of offering a delta file before sending the complete delta file is primarily to save network traffic. It also helps us satisfy M.1 of reliable multicast since devices will then not apply delta files that have already been applied. As a device can receive multiple offers for the same delta file, it will only accept delta files that it has not already received. The use of locks makes sure that offers for the same delta file are only rejected once the file has been delivered. This helps us satisfy M.1 by only delivering the message once for every process. At the same time, M.2 is met since once a delta file is offered, it will eventually reach the device and be applied, either through this device or some other. By using *ack*s to confirm that a delta file has been received and sending devices delta files that they have not acknowledged, we increase the M.3 satisfaction even though we cannot guarantee M.3. Since devices can come and go on the network, it cannot be ensured that all devices will receive the delta files, so to correct for that, we introduce delta file exchange.

Delta File Exchange

A device sends an *offer_exchange* event to all trusted devices once the application starts up, containing an array of objects containing the file path, version, hash, and whether it is a directory or not. This tells the other devices that it has started up and wishes to catch up. During the time the device is offline from the rest of the network, two things can happen: files can be changed on the offline device, or files can be changed in the online network. This means that when a device comes online, it has to reach consensus with the rest of the network before it can apply new changes from other devices. We do that with the *delta file exchange*, which is modeled in Figure 7.7.

Again, we are using *Device A* as the sender and *Device B* as the receiver of the offer. This means that *Device A* is the device that has just connected to the network. When *Device B* receives the event, it checks if it has any delta files for *Device A*, meaning that files changed on *Device B* while *Device B* was offline. We do this by querying the *delta_file* table for all delta files that do not have an acknowledgment from *Device A* and selecting the oldest for each stored file. If there are no delta files for *Device A*, each received offer is with an *accept_deltafile* event.

However, if *Device B* has new delta files for *Device A*, the list of offers are compared to the list of the fetched delta files, to find out if there are any clashing delta files. Two delta files are clashing if their path and version are equal. If any clashing delta files's hashes are equal, meaning it is the same file, we send an acknowledgment back to *Device A*. In case the hashes are not equal, we emit a *conflict* event to *Device A* with the file location, version, as well as our delta file serialized in JSON format. Whenever the conflict regards a directory, the two directories are simply merged, and an acknowledgment is sent back. For every offer that is not conflicting we emit an *accept_deltafile* event. For every delta file for *Device A* that is not conflicting, we emit an *offer_deltafile* event.

By offering devices delta files that they have not yet acknowledged, we come closer to satisfying item M.3 of reliable multicast, namely that *"If a correct process delivers a message, then all other processes in the group will eventually deliver the same message"*.

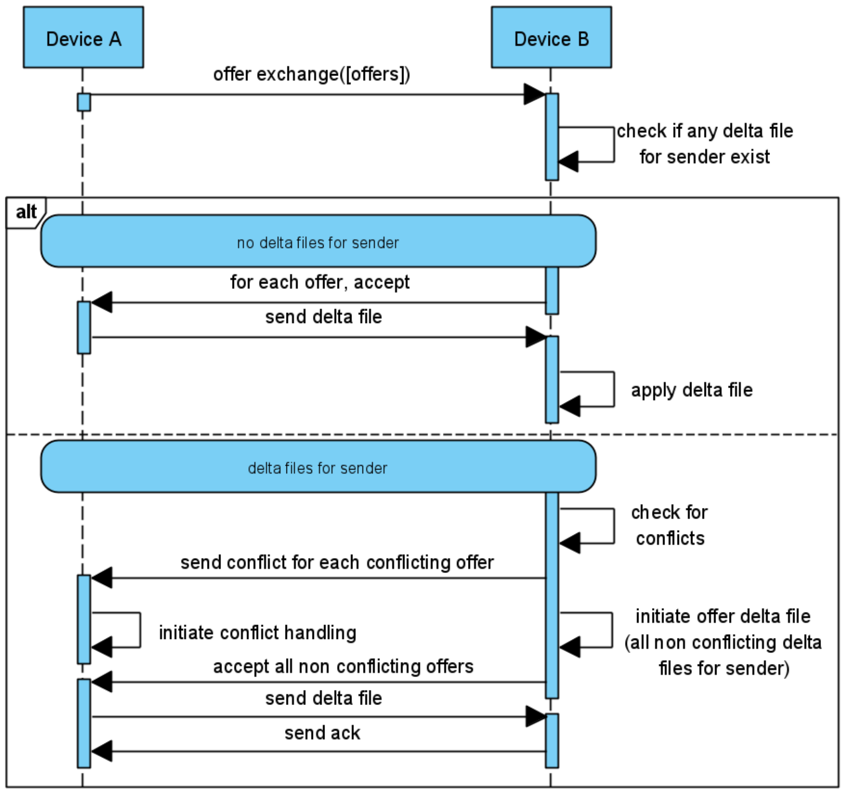

Device A
Device B
offer exchange([offers])
check if any delta file for sender exist
alt
no delta files for sender
for each offer, accept
send delta file
apply delta file
delta files for sender
check for conflicts
send conflict for each conflicting offer
initiate conflict handling
initiate offer delta file (all non conflicting delta files for sender)
accept all non conflicting offers
send delta file
send ack

Figure 7.7: An UML sequence diagram showing the flow of a delta file exchange between 2 devices.

Conflict Handling

As described in the previous sections, a conflict can happen whenever two different delta files of the same version try to affect the same file. Whenever a conflict happens, it is the responsibility of the sending device to handle the conflict. In the subsequent paragraphs, this is *Device A*. The sequence of conflict handling between two devices is shown in Figure 7.8, and the activities of the responsible device are shown in Figure 7.9.

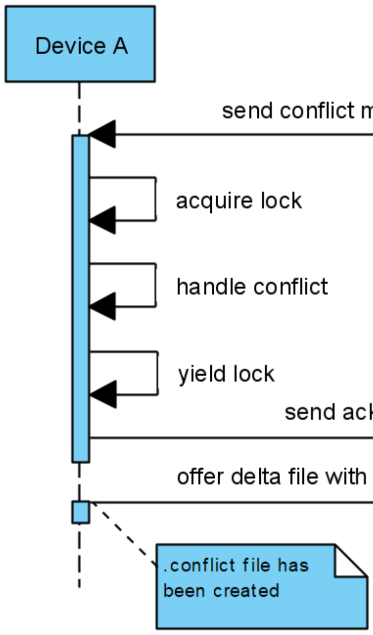


Figure 7.8: An UML sequence diagram showing the flow of a conflict handling between 2 devices.

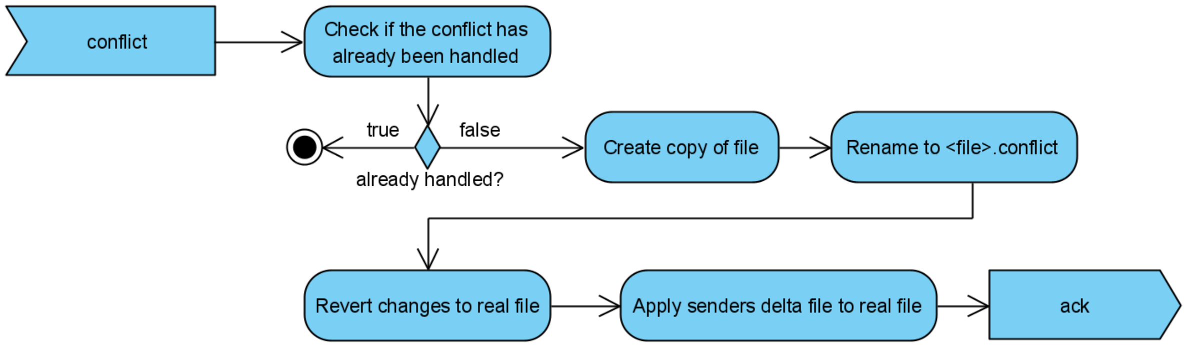

conflict
Check if the conflict has already been handled
true
false
already handled?
Create copy of file
Rename to <file>.conflict
Revert changes to real file
Apply senders delta file to real file
ack

Figure 7.9: Conflict handling on the responsible device.

When *Device A* receives a *conflict* event, it tries to acquire an internal lock identified by the file path. Once the lock is acquired, it checks if the conflicting delta file has already been received, by querying the delta file table for that file location and version, and checking if the result is equal to the delta file received. If that is the case, the conflict is already handled, and the event is ignored. This can happen if two devices send *conflict* events at the same time on the same delta file. If it has not been handled, we copy our current file that is conflicting and saves it with the `.conflict` file extension. When the file has been copied, we revert our changes to the original file by reverting the delta files we have stored. After the file has been reverted, the incoming delta file is applied, and the old delta files are erased. Because the conflict file was created, a new delta file is created that is sent out to the trusted devices. At the end, an acknowledgment from the sender is stored for the delta file, the lock is yielded, and an acknowledgment is sent back to *Device B*.

7.4.4 Asynchronous Locks

We are using the NPM package ASYNC-LOCKS to implement locks. This makes sure that only one instance of code can execute at once, when the instance has acquired the lock. Alongside the library, we create a map that maps lock keys to functions that allow the lock to be yielded. We do this to allow the *send_deltafile* code to release locks acquired in *offer_deltafile*. The Code Snippet 7.4 shows how the locks are acquired and saved in the map.

```
1 lock.acquire(storeFilePath, async (done) => {
2   aquiredLocks.set((id+storeFilePath), () => {
3     aquiredLocks.delete(id+storeFilePath);
4     done();
5   });
6   // The code to be executed after acquiring lock
7 }
8
9 const lockRemover = aquiredLocks.get((id+deltaFile.fileLocation));
10 lockRemover();
```

Code Snippet 7.4: Code snippet of the locking code

First, we acquire the lock by calling `lock.acquire` with the key of the lock. Once we have acquired the lock, we update the map with a function that deletes the element from the map and release the lock via the `done` function. The `done` allows for the explicit yield of a lock. We can then fetch the function by calling `aquiredLocks.get` with the key of the lock, as seen in line 9, and then called as in line 10. If we do not include the done function when acquiring the lock, the lock is simply yielded when the function returns.

The reliable multicast protocol that our solution is based upon, could be implemented directly if the need should arise. Having implemented the protocol, we could keep the three properties *Integrity*, *Validity* and *Agreement*. We do not implement the multicast protocol in order to have a higher usability of the system. If the *Agreement* property is to be kept, the user would not be able to see their changes, as the system must be able to guarantee that the changes is delivered to all other devices. A possible implementation could be reworking the storage system, so that each device has their own personal storage and a shared storage, that tried to reach consensus with the trusted devices. The protocol we implement has some flaws that could be solved by using more suitable protocols for our application, mainly in relation to our relaxation of the *Agreement* property.

8 | Graphical User Interface

In order to fulfill requirement **R5**, we design a user-friendly GUI to increase our application's usability. The GUI connects the user and the backend of the system and presents the functionality that the user can interact with. As a guideline, we draw inspiration from the principles of GUI design presented by David Benyon in Designing Interactive Systems [38]. He presents the Gestalt Laws which are based on how the innate human perception works [38, p. 557-559]. To accommodate these laws, we will present our functionality to the user in the layout depicted in Figure 8.1.

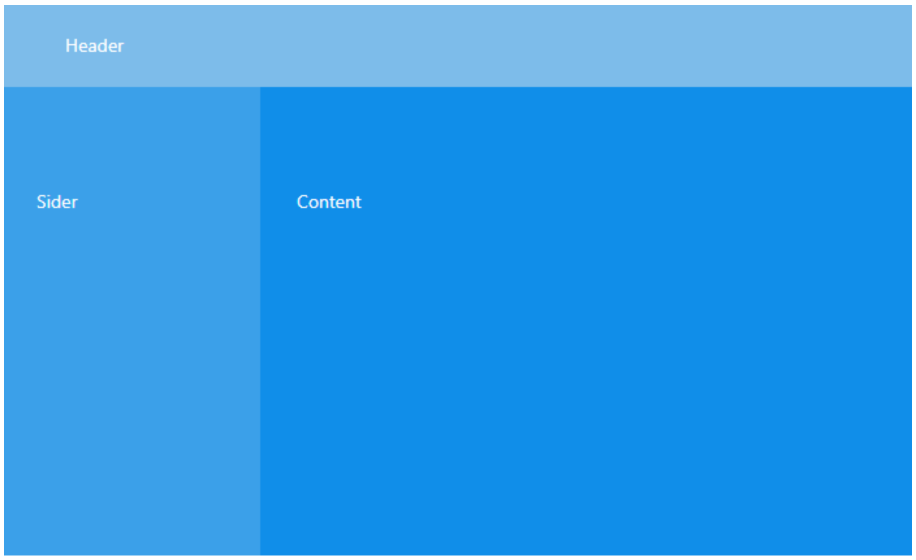
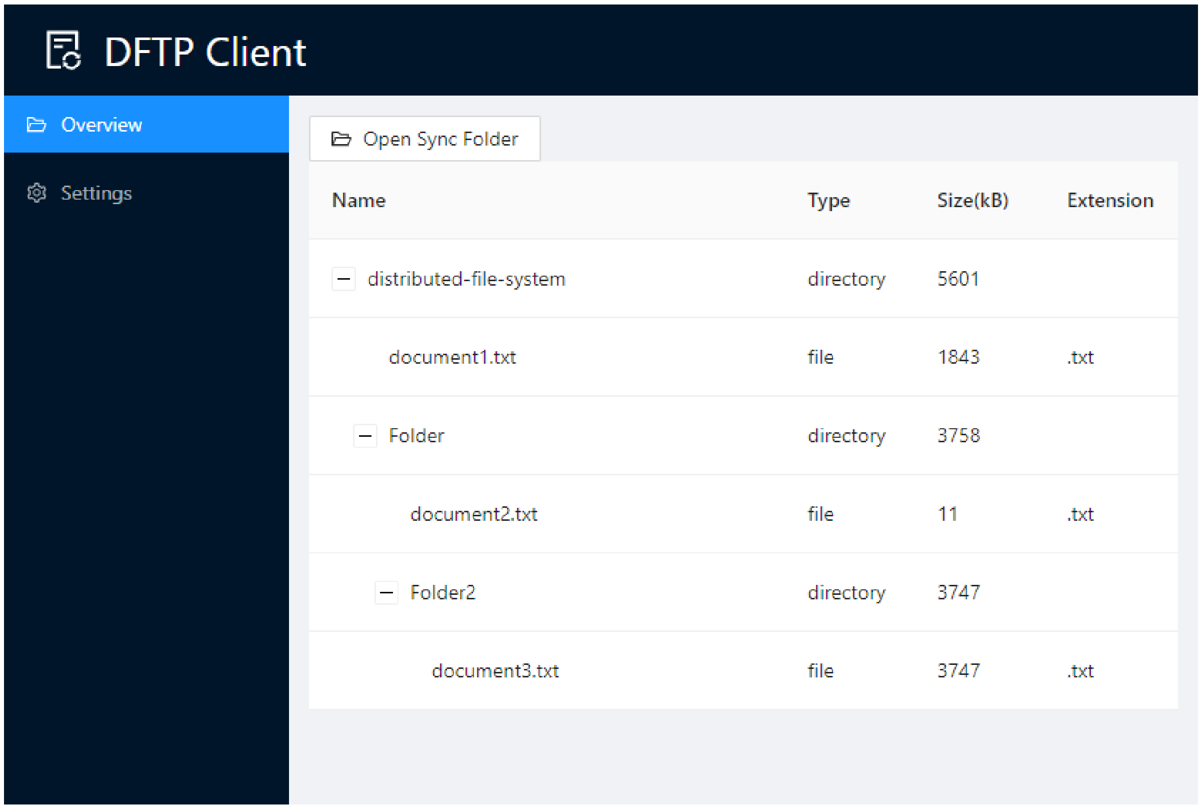


Figure 8.1: The GUI layout's three areas: Header, sider, and content.

By grouping elements within the header, sider, and content area of the GUI, we conform to the proximity and similarity laws, which describe that similar objects that appear close together are often perceived as belonging together. The sider area will function as a menu to allow the user to select what is shown inside the content area. The user's choices are to see what files and folders are currently included in their specified synchronized folder, who their trusted devices are, as well as their outgoing and incoming trust requests and other settings in regards to the application. The content area's elements will also conform to the Gestalt laws.

We build our application with the React [39] front-end library together with the Electron [40] framework. Furthermore, we utilize Ant Design's [41] GUI framework in order to make use of pre-designed UI components. Electron is a framework that uses Chromium [1] and Node.js to render web applications as desktop applications [40]. This means that the application is running in a Chromium browser window as JavaScript. Since the browser does not support native modules such as SQLite, Electron can inject Node.js support into the renderer process, meaning that the application can run everything that Node.js can run. As TypeScript can run natively in neither Node.js nor the browser, the code must be transpiled into JavaScript, which is done with webpack. webpack also bundles the code and creates the HTML file importing the code [43].

Depicted in Figure 8.2 is the part of the GUI where we show the synchronization folder's directory tree to the user. In this view, all folders are collapsible to minimize clutter if the user has many files and folders. The *Open Sync Folder* button opens the synchronization folder in the computer's file browser.



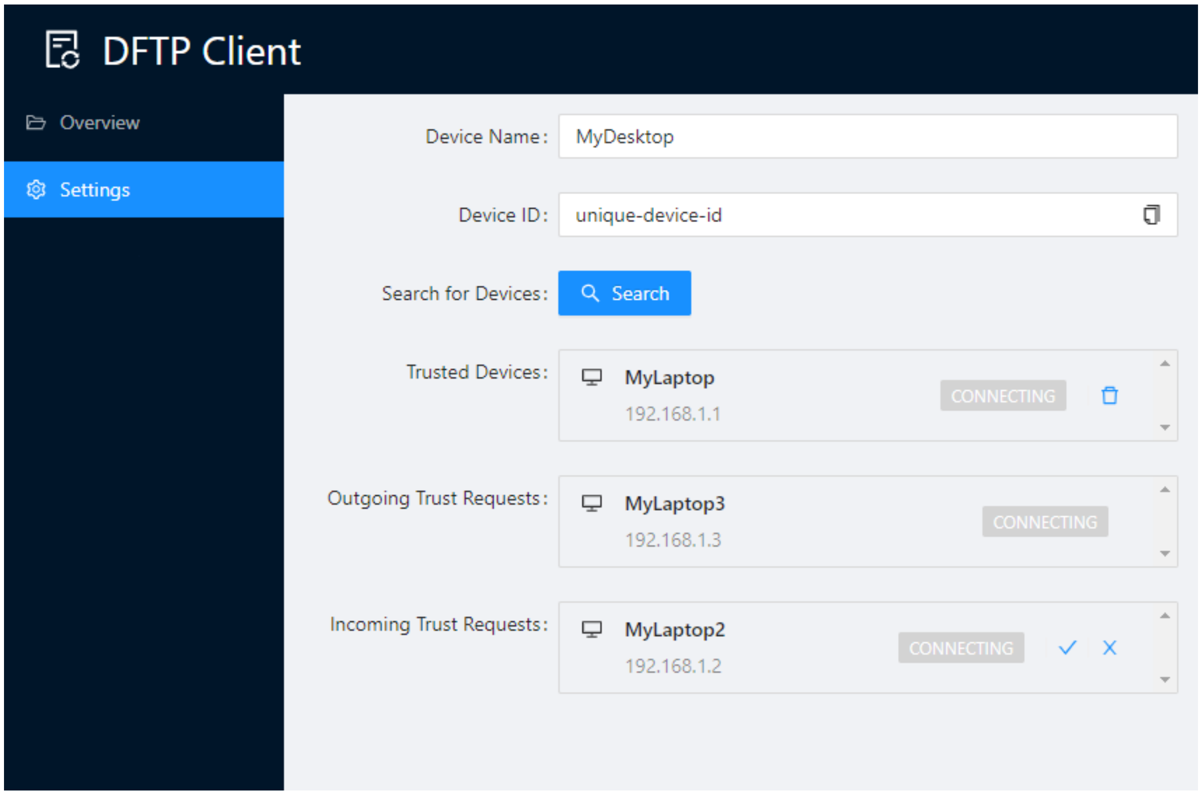
Name	Type	Size(kB)	Extension
− distributed-file-system	directory	5601	
document1.txt	file	1843	.txt
− Folder	directory	3758	
document2.txt	file	11	.txt
− Folder2	directory	3747	
document3.txt	file	3747	.txt

Figure 8.2: The *Overview* page of the application.

Figure 8.3 shows the page of the GUI, where the user can view and manage their settings, such as the device's name and trusted devices. Clicking the search button opens a pop-up that allows the user to send trust requests to any eligible device discovered on the network. Below this button, the user's

[1]Chromium is the open source project behind Google Chrome [42].

trusted devices are shown, with an option to remove a trusted device from this list. Below the list of trusted devices, the user can see both their outgoing and incoming trust requests, with the option to accept or reject an incoming trust request. If the user accepts or rejects a trust request or tries to remove a trusted device, they will be prompted for a confirmation.



Device Name: MyDesktop

Device ID: unique-device-id

Search for Devices:

Trusted Devices: MyLaptop 192.168.1.1

Outgoing Trust Requests: MyLaptop3 192.168.1.3

Incoming Trust Requests: MyLaptop2 192.168.1.2

Figure 8.3: The *Settings* page of the application.

The effect of accepting a trust request from a device that is a member of a trust group will conform to the trust rules exhibited in Section 7.3. That means all members of that group will also be added to the user's trusted devices. Because of that, the trusted devices of the requesting device can be displayed by hovering over the trust request, as shown in Figure 8.4.

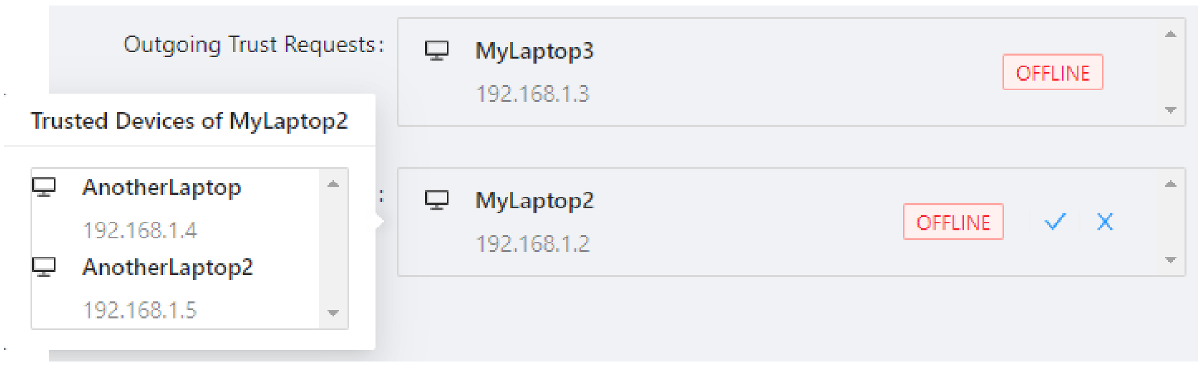


Figure 8.4: An example of a trust request with another trust group.

We assume that our application provides a usable interface between the user and our system's back-end. However, this assumption is solely based on our guidance of the GESTALT LAWS. To clarify whether we are right or wrong and find a correct measurement of the GUI's usability, we can conduct usability tests with external test subjects. This will provide even greater insight, as we can get feedback from users with different backgrounds and technical skills, as the intended end user might not be exclusive to software engineering students. However, usability tests are outside the scope of the scope of the project.

9 | Testing & Evaluation

We perform a number of system tests to measure performance and scalability. This chapter presents the results and evaluation of those tests. All tests are done manually, except parts of the performance tests, which are conducted by an external program as described in Section 9.3. Each section describes a list of tests followed by the test results and describes any potential sources of error.

9.1 Feature Tests

In this section, we present a series of tests based on the requirements described in Section 3.1.2. We perform each test manually on two and three devices, and the results are shown in Table 9.1. We conduct all tests with an arbitrary file of only a few bytes in size. Since these tests do not measure performance but rather functionality, we do not consider it important to use the same files for each test.

Prior to each test, we reset each device by clearing their database and all files on the disk, meaning all devices involved in the system will start in a state of consensus. In addition to the system being cleaned, the devices involved only have each other as trusted devices. Each test results in either *Success* or *Failure*. *Success* denotes that the system ends in a state of consensus, with all files and folders having the correct content and the list of trusted devices is the same on all devices. *Failure* denotes that we do not reach consensus in the system, with one or more errors in the files, folders, or lists of trusted devices.

Three of the tests in Table 9.1 fail, namely B11, B13, and B14. Certain challenges occur when events are triggered at the exact same time on different devices. A problem arises when multiple users make edits to the same document simultaneously and the system will not handle this synchronization, thus not reach consensus.

When a user sends a trust request to another device, the application will first check if it has an incoming request from that device. If the device is already in the incoming requests, then the *TrustService* will accept the request and send an acknowledgment to the other device. The failure B13 happens due to sending the requests at the exact same time and accepting them at the exact same time. This leads to the system sending too many acknowledgments and one of the devices will have itself as a trusted device.

ID	Description of test	2 Devices	3 Devices
B1	Creating a new file	Success	Success
B2	Create a new directory	Success	Success
B3	Rename a file	Success	Success
B4	Rename a directory	Success	Success
B5	Making a 5-25 byte change to file content	Success	Success
B6	Making a 1 Kb change in a single line	Success	Success
B7	10 paragraphs of lorem ipsum in a new file	Success	Success
B8	Moving a file	Success	Success
B9	Moving an empty folder	Success	Success
B10	Moving folder with files	Success	Success
B11	Saving conflicting changes simultaneously on all devices	Failure	Failure
B12	Establish trust between device(s), one at a time	Success	Success
B13	Both devices sending a trust request to each other at the same time	Failure	Failure
B14	Send all files when two devices establish trust	Failure	Failure

Table 9.1: This table shows all tests performed with two and three devices involved in the system and their results.

The implementation does not handle sending files to all new devices when trust is established between devices. Therefore, B14 fails at this point in time, and a mechanism to multicast to the new devices has to be created for this functionality to take place.

9.2 System State Tests

In this section, we list tests of different states that our system and devices are prone to experience in a live environment. We use the same definition of *Success* and *Failure* as defined in Section 9.1.

States with Two Devices With two devices in the system, availability is an apparent factor. Both devices can be online, resulting in each time one makes changes, the other device will immediately receive the change. If we let one of the devices go offline, they will both be able to make changes, but they will never reach consensus until they are both online at the same time. When the offline device comes online, they will both offer the other their files and reach consensus. If both devices are offline, they will handle it similarly to when one of the devices is offline; the last to come online will contact the other device. We present the results of the tests in Table 9.2.

ID	Description of system state	Result
2D1	One device is online and the other is offline. The offline device comes online, and they have a conflict when they created the same folder.	Success
2D2	One device is online and the other is offline. The offline device comes online, and there are no conflicts.	Success
2D3	One device is online and the other is offline. The offline device comes online and there are conflicting files.	Success
2D4	Two devices offline. Both devices makes a conflicting change to the same file and come online.	Success
2D5	Two devices offline. Both devices makes a change that does not conflict. Both devices come online.	Success

Table 9.2: This table shows the tests performed with two devices when some of them are offline and their results.

States with Three Devices When we include three devices in the system, our system is prone to a new list of failures. When all the devices are online, they are all able to make and receive changes from each other. If one of the devices goes offline, the other two devices will keep sending delta files between each other and reach consensus between them. Thus, we will never see the system reach consensus until the last device comes online again. When the last device comes online, we may see some conflicting files that require the previously offline device to roll back to the last point they reached consensus and apply their changes instead. The device will also move its changes to a conflict file. When we have two of the devices offline, none of the three devices will reach consensus. As soon as one of the offline devices comes online, the two online devices will try to handle conflicts and reach consensus. We see this process repeated until the last device comes online. We can exhibit numerous scenarios between multiple devices, and we present some of the more common scenarios in Table 9.3.

ID	Description of system state	Result
3D1	Two online devices. One offline device. The online devices creates a new file. The offline device comes online without any conflicting files.	Success
3D2	Two online devices. One offline device. The offline device creates a file, that does not conflict with any file on the online devices and comes online.	Success
3D3	Two online devices. One offline device. The two online devices have consensus. The offline device creates a conflicting change and comes online.	Failure
3D4	One online device. Two offline devices. All three devices make conflicting changes to the same file and one of the offline devices comes online. After the conflict has been handled, the other offline device will come online.	Failure

Table 9.3: This table shows the tests performed with three devices when some of them are offline and their results.

3D3 will reach consensus if the three devices first reach censuses before the devices goes offline. As it comes online with a conflicting change, the device will handle the conflict and send it to the trusted devices. However, if the device goes offline and creates a conflicting change on a file that has not previously been reached consensus on and comes online, the "real" file will not have any content, while the .conflict will be correct.

An error occurs in 3D4 when multiple conflict files have to be sent to the network. Initially the online device and the first device coming online create consensus. The device coming online creates a conflicting file and cascades the conflict file through the network. The issue arises when the third device comes online with a conflicting change. It will create consensus on the "real" file, but it is unable to send the .conflict file to all devices in the network, as they have a previous .conflict file with that version already.

9.3 Performance

In this section, we conduct two kinds of tests on the system and present our methodology and results. Our first test explores how long it takes for a small change in a file to synchronize between devices, while the second explores how much time it takes for increasingly larger files to be synchronized. All test results, and the testing program used, can be found in `testing.zip`, which follows this report.

All tests are performed on the devices presented in Table 9.4 connected either through WiFi or ethernet cable to the same local area network. The test network has an ethernet speed of 100/100 Mbps and a WiFi speed of approximately 35/65 Mbps.

ID	OS	Processor	RAM	Storage	Networking	Form factor
A	Windows 10	Ryzen 7 3700X (8 cores)	32 GB	SSD	Ethernet cable	Desktop
B	Windows 10	Intel i7 7500U (2 cores)	16 GB	SSD	WiFi	Laptop
C	Windows 10	Intel i7 8750H (6 cores)	16 GB	SSD	WiFi	Laptop
D	Windows 10	Intel i7 6820HQ (4 cores)	8 GB	SSD	WiFi	Laptop

Table 9.4: An overview of the hardware configuration of the different devices used for testing.

9.3.1 Synchronization Time

To test the system's performance, we take inspiration from the concept of a token passing in a ring topology network [44, p. 123-124]. Token passing is the principle of a networked device that must wait until it has received a token to communicate with other devices. In a ring topology network, each device can only receive messages from the device prior to it and only send them to the one after it. This results in the devices being ordered in a ring with the flow of information going in a single direction.

The concept of token passing will provide the system's performance tests with a form of access control by restricting only one device to write to a file at a time. By arranging the devices in a ring topology, we can measure how much time it takes for changes to arrive at a device. The access control provided by token passing will make sure changes only happens on a single device.

Testing Program

To log our test results, we create an external program to monitor the changes made by each device. Our testing program will continuously read from the synchronized file and wait for a character denoting the previous device in the sequence to be written. When it reads that character, it will write a timestamp followed by its own specific character that the subsequent device's testing program is waiting for. We configure the testing program such that only one device in the trusted group reads and looks for each unique character. Hence, we will achieve a system behavior approximate to as if configured as a token passing ring topological network. We see our configuration of three devices in Figure 9.1. The code of the testing program can be found in `testing.zip`, in the file `main.py`.

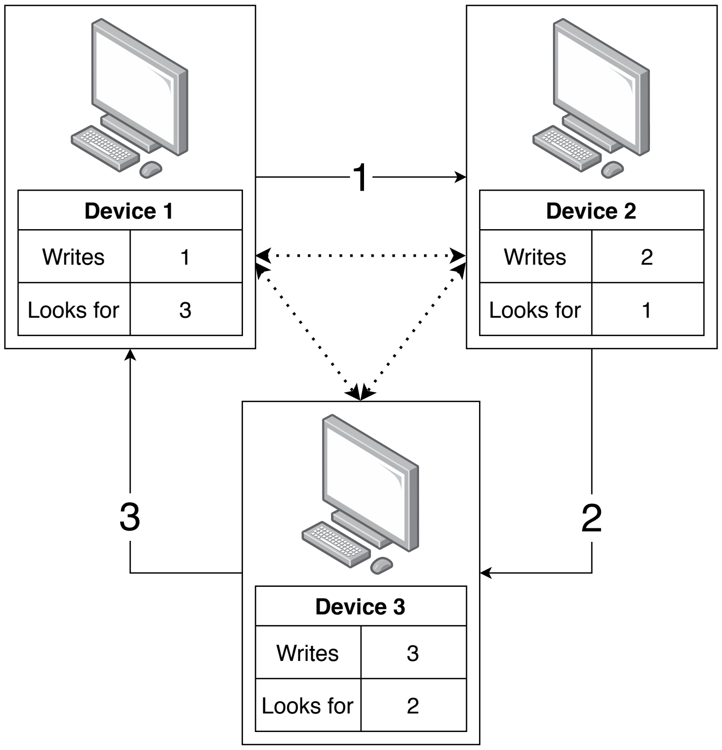


Figure 9.1: Our configuration of the testing program involving three devices. If we only use two devices, they will continuously read for the character of the other device. Using the dotted lines, we indicate the importance of remembering that all devices communicate with each other and continuously try to reach consensus regarding the file content. However, we can only let one device write to the file at a time, as we dictate by the testing program.

The size of the changes we are monitoring is relatively small, ranging from 19 to 22 bytes, with 22 being most common. These changes consist of a timestamp separated by a device's ID. The timestamp varies from 15 to 18 bytes, with a separator and the ID of the device taking two more bytes. The last two bytes come from the newline written at the start of a write, which is represented as "\n" in text.

Test Procedure

We perform our tests by having two, three, and four devices, all being members of the same trust group. In our testing program, *Device 1* will write a timestamp followed by a "1" in the file whenever it sees a "2" at the end of the file. *Device 1* is responsible for creating the file initially. Therefore, it writes a "1" in the file when the testing program is started. *Device 2* will write a "2" in the file, and a timestamp, whenever it sees a "1". This setup results in the file alternately being edited by *Device 1* and *Device 2*. The tests with three and four devices are set up in a similar manner, as illustrated in Figure 9.1.

We run our tests with 50 iterations for each device with a delay of 1 second between each check of the file's content. Since our total number of writes is dependent on the number of devices in the system, our test with two devices has 100 iterations, three devices has 150 iterations, and four devices has 200 iterations total. Our two-device test is conducted on devices A and B from Table 9.4. We use A, B, and C for our three-device test, while we use all four devices for the last test.

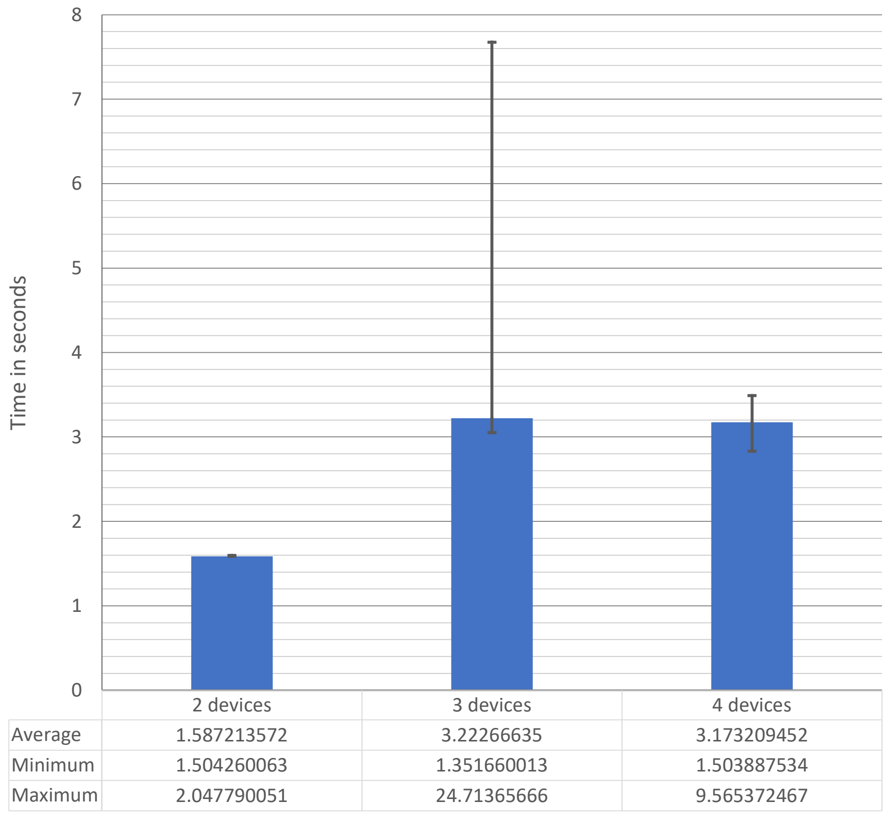
Results

In Figure 9.2, we give an overview of our test results where we present our timings in seconds The calculation of the tests use all the timestamps made by our devices. We find our complete data set in `testing.zip`, in the files `2devices.txt`, `3devices.txt`, and `4devices.txt`.

Since the system clocks across multiple devices are often not synchronized , we must account for this in our test. First, we take all timestamps from a single device and calculate the differences between them. We then know how long it takes each writing cycle to get through the ring topology network and back to the device, defining a round trip time. Lastly, each result is divided by the number of devices in the group to get an average of how long each device takes to transmit the message to the next, for the specific round trip. This is done for all devices in the group and we then calculate the average of all these values. The minimum and maximum values are drawn from the same averaged round trip times.

Sources of Error

Our testing program may be prone to errors within the one second delay, as we are unable to register any changes made by a device during it. In the worst-case, it will take exactly one second, which will be reflected in the timestamp. In order to mitigate this, we run all tests with the same delay. We observe another source of error when reading and writing to the file. In order to reload the logging file, our testing program closes the file when it is done checking and then reopens it after the one second delay to look for new changes. This can introduce further delay to our test. However, since all our devices run on an SSD, as presented in Table 9.4, we consider this delay to be negligible.



	2 devices	3 devices	4 devices
Average	1.587213572	3.22266635	3.173209452
Minimum	1.504260063	1.351660013	1.503887534
Maximum	2.047790051	24.71365666	9.565372467

Figure 9.2: The blue bars show the average time it takes to synchronize a change relative to the number of devices involved in the test. We mark the standard deviation as a black bar on top of the average time. We calculate it in a "positive" and "negative" part, with all data points above the average contributing to the "positive" standard deviation and all data points below the average contributing to the "negative" standard deviation. The table below the bars shows the average, the minimum, and the maximum time it takes for a change to synchronize to the next device.

9.3.2 Character Scalability

To test how long it takes for a file to be synchronized relative to the amount of characters that are added, we conduct a test where each device will write paragraphs within a newly created file. For this purpose, we use the website `loremipsum.io` to generate different lengths of text varying between 5-45 paragraphs. The complete data set can be found in `testing.zip`, in the file `scalability.txt`. As we see in Figure 9.3, our test results are showing slight deviations. Nonetheless, we deem our coefficient of determination acceptable for estimating the time it takes to send a file depends exponentially on how large the file is.

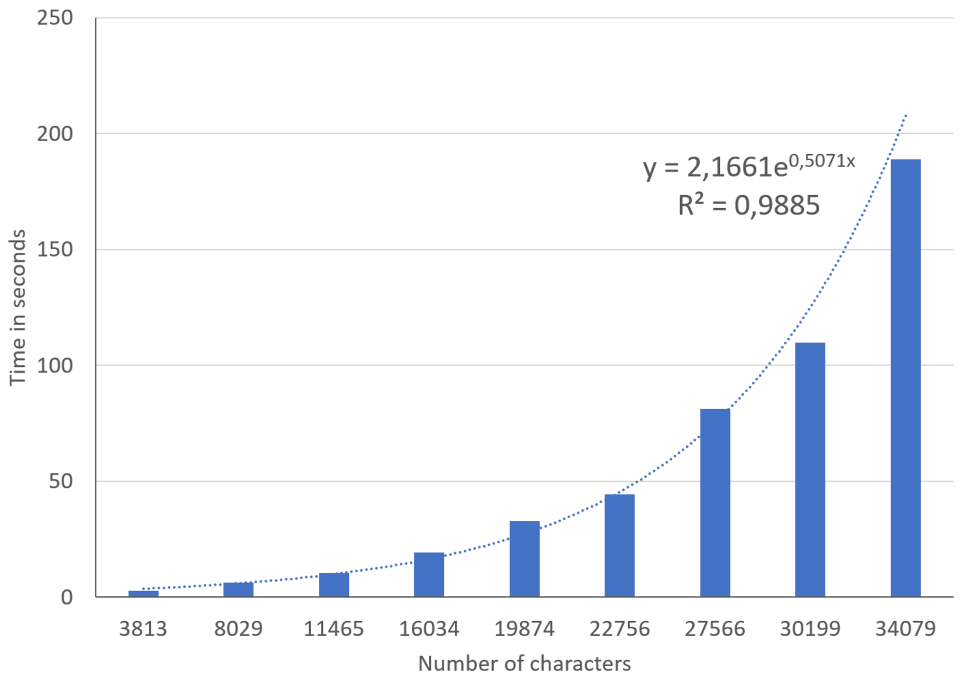


Figure 9.3: This figure shows how long it takes for our system to synchronize scales exponentially in relation to the file size.

We assume that the function we use to calculate the delta between two files is responsible for most of the run time. By way of example, we observe a run time of 167 seconds when we use the function to calculate the difference with a text consisting of 34079 characters.

9.4 Evaluation of Scalability and Performance

In order to get an idea of how the test results found in Section 9.3 relates to the real world, we do a similar test on Dropbox and Google Drive. For Google Drive, we use the same testing program as in Section 9.3 in order to have the same delay and other sources of error. Dropbox does not synchronize properly when the testing program keeps the file open, so instead, we time it manually with a stopwatch. The manual timing is a large source of error but the test should still indicate the system's effectiveness. These tests are performed between two devices.

Dropbox provides a feature called LAN Sync [45], which allows the user's device to synchronize using a LAN connection to other devices. While not explicitly mentioned by Dropbox, this should increase the performance, even though the program still needs to be connected to the internet. The test results can be seen in Table 9.5 and the complete data set can be found in `testing.zip`, in the files `2devices.txt`, `3devices.txt`, `4devices.txt`, `dropbox.txt`, and `drive.txt`.

System	**Average time**	**Minimum time**	**Maximum time**
Our system (2 devices)	1.59	1.50	2.05
Our system (3 devices)	3.22	1.35	24.71
Our system (4 devices)	3.17	1.50	9.57
Google Drive	11.90	11.02	12.69
Dropbox	3.56	3.15	4.86
Dropbox LAN Sync	3.00	2.83	3.27

Table 9.5: An overview of the systems tested. All times are in seconds and rounded to two decimals.

The results we gather in Section 9.3 shows how our system is able to scale to multiple devices, when going from 3 to 4 devices, the average round trip goes from 3.22 to 3.17, which is within an expected variance. The system can sometimes have a large averaged round trip time, as seen by the outlier in our performance test with 3 devices. We theorize this is related to the system not yielding its locks before the max timeout of 30 seconds is reached.

Our system performance is adequate when compared to commercial solutions in the use cases we have tested. When our system only uses two devices, it is much faster than both DROPBOX and GOOGLE DRIVE. This is due to the fact that it does not need to check with additional devices when a device receives a file, as it knows the sender must have the file. Likewise, the sender must only receive a single acknowledgment as it only sends to one device. We therefore compare our systems highest average, which is when using three devices, to DROPBOX and GOOGLE DRIVE. Comparing to DROPBOX, DROPBOX with LAN sync, and GOOGLE DRIVE, we see our system performing ~0.34 seconds faster, ~0.22 seconds slower, and ~8.68 seconds faster on average, respectively.

If we were to add an arbitrary large number of devices, denoted by N, each device would have to multicast to $N-1$ devices every time a change happens. This would drastically increase the amount of acknowledgments which will result in an "ack implosion" [8, p. 647]. Furthermore, sending a delta file to N devices would take an impractical amount of time, as the device has to send the file to every other device.

Of all tests, we get the most conspicuous results in Section 9.3.2, where we see an exponential growth in run-time relative to calculating the difference between two bodies of text. This is expected as the *diff*-algorithm has a $O(ND)$ run-time, but can be exponential if D is the size of N, which is true in the test we conduct. Despite it being the expected outcome, it does not reflect the desired user experience when running the system. The system is designed to be modular and each component and service can be replaced within. This test indicates that the specific module within the *DeltaFileHandlerService* is causing the program to halt, and could be swapped for a different solution. We could argue that it is not necessary to calculate differences at a character level and that it is sufficient to calculate differences at a word or even line level. Another approach could be to generate differences on line levels and then iterate through the lines and check only those containing differences. This could, of course, increase the worst case running time even further since if every line contains differences, we would first go through the lines and then every character or word.

9.5 Evaluation of Requirements

We go through each requirement from Section 3.1.2 to deem whether we have fulfilled them, hence implemented a sufficient solution to our problem statement. A full overview of each requirement and its evaluation is seen in Table 9.6.

REQ.	Evaluation	Explanation
R1	Fulfilled	We show in multiple feature tests of the system in Section 9.1 that we can create, delete, and change files and folders within the shared folder of two and three devices and reach consensus in the end. This functionality makes the basis of our distributed replicated file system.
R2	Fulfilled	Our implementation of conflict files and their creation when clashing events occur between devices ensures that we eventually reach consensus. With the conflict files, the user will not lose any data as they serve as a backup of the conflicting change's content.
R3	Fulfilled	It is only possible to synchronize files between devices that are a part of same trust group. This is achieved with our implementation of the trust sequence and acknowledgment process.
R4	Partially Fulfilled	Our protocol implements a basic architecture inspired by multicast, meaning that each device is responsible for multicasting the delta files to other devices in its trust group on the network. The current implementation only allows for storage devices. However, we present a solution to comply with our requirement of allowing devices to serve as a relay device as well. To let a device function as a relay device, we will remove its storage component.
R5	Fulfilled	A GUI is built in Electron and React and provides the user with simple elements to interact with the application's functionality, such as its trusted devices and storage folder.

Table 9.6: We evaluate each requirement from Section 3.1.2, and provide an explanation of each.

Our evaluation shows that the systems intended behavior is partially achieved with the current implementation. This is despite our evaluation of **R4**, where the absence of relay devices means decreased availability of our system. We do not believe a fulfilling solution is difficult to implement due to our current solution's modular design.

10 | Discussion

A focal point of our solution is to provide high first-party privacy, as mentioned in Section 2.1.3. This naturally transitions to high user transparency. It is easier to convince a user of our solution's high privacy if there is transparency regarding how their data is sent and where it is stored. We accomplish this by keeping the system wholly peer-to-peer, using the user's already existing devices, and giving the user full control over what devices their data is stored on. The usage of the user's own devices, i.e., consumer devices, also correlates to another focal point, namely developing an alternative to private users' existing commercial storage solutions. This adheres to the UN's 12th sustainable development goal of ensuring sustainable consumption and production patterns. We argue that this approach can reduce electricity consumption and the production of hardware needed to maintain and upgrade the commercial companies' storage solutions.

Our system manages to let users apply their own devices as storage devices and share files between them, but as discussed in Section 3.1, our system architecture also included the functionality of relay devices. These would allow the system to function as an emulated cloud service by potentially keeping a *"personal cloud"* on the user's device. In doing so, we would also gain the benefits of the commercial cloud storage solutions, explained in Section 2.2.4, which a distributed system limited to a LAN lacks. This translates to higher reliability, as the uptime would be increased if a device such as a smartphone was used as a relay device.

The testing of our solution shows room for improvement in regards to performance and consistency in specific use cases. However, as we deliberately design our system architecture with modularity in mind, possible solutions to these problems will be straightforward to implement once clearly defined.

Our solution has the foundation of a system that provides a viable alternative to existing cloud storage solutions. It does this while adhering to our problem statement in Section 2.3, which is formulated based on our analysis in Chapter 2.

11 | Conclusion

We have shown how to construct a distributed storage system for private individuals, complete with a GUI that can run on any Windows 10 device to solve the problem we propose in Section 2.3. A benefit of using a distributed architecture means that our solution is not dependent on a centralized server to synchronize files. This architecture also allows us to create an application that maintains high privacy, user transparency, and reliability. Improvements to privacy and user transparency stem from a user's ability to explicitly choose which of their own devices should synchronize their files. This also aids in solving the problem of utilizing unused storage space by providing the opportunity to use already existing hardware. To sustain these two user-needs when connecting multiple users, we introduce the notion of trust, which allows for groups of devices to synchronize files between them. The system's level of reliability is dependent on how many devices a user has interconnected. Having more devices increases the likelihood that the most recent file versions are available when needed. On the other hand, more devices also mean an elevated number of conflicting changes in the system, which in turn increases the risk of a consensus issue.

We introduce the concept of storage and relay devices. Storage devices are the devices from which the user can make changes to the files and receive file changes. Relay devices are responsible for propagating file changes to other storage and relay devices. While we ultimately did not implement the option to distinguish a storage device from a relay device, a clear distinction between these two devices' features allows for a more modular design of our system architecture. We achieve this by isolating functionality to a storage component and a relay component. Storage devices utilize both components, while a relay device only utilizes the latter. This illustrates how a relay device's functionality can be changed relatively straightforward.

We tested the system's features and evaluated the results, which show that most operations, such as adding files or folders, moving, and deleting them, works as intended. In some scenarios, however, the system does not behave as it should, e.g., if two devices create files simultaneously or devices with conflicts come online simultaneously.

We have also tested the system's performance and compared it to the commercial storage solutions Dropbox and Google Drive. Our system has a much higher maximum synchronization time than both Dropbox and Google Drive, and as discussed in Section 9.3.2, the time spent calculating file changes seem to increase exponentially in relation to the size of the changes. However, when comparing the average time for synchronization to happen, our system is only ~0.22 seconds slower than Dropbox with LAN sync enabled when using three devices, while it is ~0.34 and ~8.68 seconds faster than Dropbox without LAN sync and Google Drive, respectively. Using only two devices, it outperforms Dropbox with LAN sync, Dropbox without LAN sync, and Google Drive by 10.31, 1.97, and 1.41 seconds, respectively.

12 | Project Management

This chapter will present the different methods used for managing the project. What methods to use were quickly agreed upon within the project group since all group members had used similar working processes in earlier projects.

Overall Planning

We created an overall plan for the project using the waterfall model, containing deadlines for every significant part of the report and program. Designing the overall plan using the waterfall model helped ensure we reached our goal within the given time frame. We imported the overall plan into an online calendar, that all group members had access to, which provided an increased overview. We also used this calendar to schedule personal events not related to the project, e.g., doctor appointments, so that the group member would be absent during scheduled project work.

As the larger milestones were specified in the overall plan using the waterfall model, we used a Kanban-like approach to distribute the workload among the group. This helped ensure rotation between different areas of the project, which helped make sure that not just a single group member had all knowledge on an area of the project. This approach also mitigated the feeling of working in the same area for too long and feeling stuck. Furthermore, it improved motivation since each group member could choose what task they would most like to work with, given that it was not already taken. The smaller tasks used in the Kanban-like approach, made from the larger milestones, were organized in the online tool TRELLO. This worked well since all group members had frequently used TRELLO in previous projects. For TRELLO to have been more effective during the project, it should have been updated more frequently than it was. Otherwise, it functioned as intended.

Daily Work

Over the course of the project, each day started with daily Scrum. This was chosen since the project's domain was relatively unknown. A daily update from the other group members helped us get a clearer picture of the progress and distribute knowledge evenly among the group. It also facilitated discussion about various areas of the report and code, if other group members had ideas for improvements, or alternative ways to accomplish something. The daily scrum might have contributed to Trello not being updated as often as it should have been. When the group already had a clear picture of what tasks were important to complete during our daily meeting, Trello might have seemed redundant. However, there were occasionally the need to use more time to update it, since it provided a nice overview when weekly smaller milestones were reached.

Besides meeting at the start of each day, we had a meeting half way through the day and at the end of the day. This also contributed to distribution of knowledge and to quickly adapt if we ran into problems. This may also have contributed to Trello not being updated. During these meeting, there were normally one or two group members who took charge of the meeting, making sure that everyone had something to continue with and no problems were left without discussing them.

Pair programming was also used in order to distribute knowledge in the group evenly. It was not only done for programming but also for writing some parts of the report where it fit. This worked well by providing those weak in one area with the strengths of others. Early in the semester, we choose to work from home because of the current pandemic. In need of an online communication platform, we chose Discord. Discord is easy to setup and every group member was already using Discord.

Throughout the design and implementation, we have taken inspiration from model driven development. This helped us implement the system more efficiently, as it provided us with clear overview of the system's intended functionalities before we started implementing them. Most of the models in this report has been made with Visual Paradigm. When implementing, we used GitHub for version control. This allowed us to share and work on the implementation at the same time while keeping track of changes.

Summary

To sum up, most methods used during the project worked well. An overall hybrid method was used, with elements from the waterfall approach, Kanban and Scrum. The daily Scrum might have collided with the frequent updates needed for Trello used in Kanban, but otherwise, the methods worked together well.

Bibliography

[1] Statista Inc. *Value of the data economy in the European Union (EU) and United Kingdom from 2016 to 2020 and in 2025* (in billion euros)*. Accessed: 2020-09-28. URL: https://www-statista-com.zorac.aub.aau.dk/statistics/1134993/value-of-data-economy-eu-uk/.

[2] US Senate. *S.2383 - CLOUD Act*. Accessed: 2020-09-28. URL: https://www.congress.gov/bill/115th-congress/senate-bill/2383/text.

[3] Dropbox. *Effektiv online storage til alle dine filer*. Accessed: 2020-09-28. URL: https://www.dropbox.com/features/cloud-storage.

[4] Statista Inc. *Forecast number of personal cloud storage consumers/users worldwide from 2014 to 2020 (in millions)*. Accessed: 2020-09-28. URL: https://www-statista-com.zorac.aub.aau.dk/statistics/499558/worldwide-personal-cloud-storage-users/.

[5] United Nations. *Goal 12: Ensure sustainable consumption and production patterns*. Accessed: 2020-09-30. URL: https://www.un.org/sustainabledevelopment/sustainable-consumption-production/.

[6] Anders S. G. Andrae. "On Global Electricity Usage of Communication Technology: Trends to 2030". In: (2015), pp. 117–157.

[7] Nigel Bevan. "Quality in use: Meeting user needs for quality". In: *Journal of Systems and Software, Volume 49, Issue 1*. 1999, pp. 89–96. DOI: 10.1016/S0164-1212(99)00070-9. URL: http://www.sciencedirect.com/science/article/pii/S0164121299000709.

[8] George Coulouris et al. *DISTRIBUTED SYSTEMS Concepts and Design*. Fifth Edition. Addison-Wesley, 2012.

[9] Alain Abran et al. "Usability meanings and interpretations in ISO standards". In: *Software quality journal* 11.4 (2003), pp. 325–338.

[10] Iulia Ion et al. "Home is Safer than the Cloud! Privacy Concerns for Consumer Cloud Storage". In: *Proceedings of the Seventh Symposium on Usable Privacy and Security*. SOUPS '11. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2011. ISBN: 9781450309110. DOI: 10.1145/2078827.2078845. URL: https://doi.org/10.1145/2078827.2078845.

[11] Inc. Amazon Web Services. *AWS Customer Success*. Accessed: 2020-09-28. URL: https://aws.amazon.com/solutions/case-studies/.

[12] Idilio Drago et al. "Inside Dropbox: Understanding Personal Cloud Storage Services". In: 2012. DOI: 10.1145/2398776.2398827. URL: https://doi.org/10.1145/2398776.2398827.

[13] Subash Chandra Yadav and Sanjay Kumar Singh. *Introduction to Client Server Computing*. First Edition. New Age International Ltd, 2009.

[14] Nathan Sebastian. *Usage & Trends of Personal Cloud Storage: GoodFirms Research*. URL: https://www.goodfirms.co/resources/personal-cloud-storage-trends. (Accessed: 2020-12-19).

[15] The Guardian. *Turkey blocks Wikipedia under law designed to protect national security*. Accessed: 2020-09-23. URL: https://www.theguardian.com/world/2017/apr/29/turkey-blocks-wikipedia-under-law-designed-to-protect-national-security.

[16] John Kubiatowicz et al. "OceanStore: An Architecture for Global-Scale Persistent Storage". In: (2000).

[17] UC Berkeley Computer Science Division. *The OceanStore Project*. Accessed: 2020-09-23. URL: https://oceanstore.cs.berkeley.edu/.

[18] Syncthing. *Welcome to Syncthing's documentation!* Accessed: 2020-09-23. URL: https://docs.syncthing.net.

[19] InterPlanetary File System. *IPFS powers the Distributed Web*. Accessed: 2020-09-28. URL: https://ipfs.io/.

[20] InterPlanetary File System. *Learn how to build the future of the internet*. Accessed: 2020-09-28. URL: https://docs.ipfs.io/.

[21] Inc. npm. *About npm*. URL: https://docs.npmjs.com/about-npm. (Accessed: 2020-12-17).

[22] TypeORM. *TypeORM - Amazing ORM for TypeScript and JavaScript (ES7, ES6, ES5)*. Accessed: 2020-12-13. URL: https://typeorm.io/#/.

[23] MDN contributors. *Promise - JavaScript | MDN*. Accessed: 2020-12-13. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

[24] Gordon Everest. "BASIC DATA STRUCTURE MODELS EXPLAINED WITH A COMMON EXAMPLE." In: (Oct. 1976).

[25] SQLite Consortium. *SQLite - 35% Faster Than The Filesystem*. URL: https://www.sqlite.org/fasterthanfs.html. (Accessed: 2020-12-18).

[26] Paul Miller. *Chokidar*. https://github.com/paulmillr/chokidar. 2020.

[27] Eugene W. Myers. "An O(ND) Difference Algorithm and Its Variations". In: *Algorithmica* 1 (1986), pp. 251–266.

[28] J. W. Hunt and M. D. Mcilroy. *An algorithm for differential file comparison. Computer Science*. Tech. rep. 1975.

[29] Kevin Decker. *jsdiff*. https://github.com/kpdecker/jsdiff. 2020.

[30] Robert Elder. *Myers Diff Algorithm - Code & Interactive Visualization*. URL: https://blog.robertelder.org/diff-algorithm/. (Accessed: 2020-12-18).

[31] MDN contributors. *Base64*. URL: https://developer.mozilla.org/en-US/docs/Glossary/Base64. (Accessed: 2020-12-17).

[32] MDN contributors. *Concurrency model and the event loop*. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop. (Accessed: 2020-12-17).

[33] Dylan Piercey. *local-devices - npm*. URL: https://www.npmjs.com/package/local-devices. (Accessed: 2020-12-18).

[34] MDN contributors. *Web Workers API - Web APIs | MDN*. Accessed: 2020-12-14. URL: `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API`.

[35] Damien Arrachequesne. *Socket.IO Documentation*. URL: `https://socket.io/docs/v3`. (Accessed: 2020-12-14).

[36] S. Floyd et al. "A reliable multicast framework for light-weight sessions and application level framing". In: *IEEE/ACM Transactions on Networking* 5.6 (1997), pp. 784–803. DOI: `10.1109/90.650139`.

[37] R. Rivest. *The MD5 Message-Digest Algorithm*. URL: `https://tools.ietf.org/html/rfc1321#section-3.4`. (Accessed: 2020-12-16).

[38] David Benyon. *Designing Interactive Systems: A comprehensive guide to HCI, UX and interaction design*. Third Edition. Pearson Education Limited, 2014.

[39] React. *React*. Accessed: 2020-12-11. URL: `https://reactjs.org/`.

[40] Electron. *Electron*. Accessed: 2020-12-11. URL: `https://www.electronjs.org/`.

[41] Ant Design. *Ant Design*. Accessed: 2020-12-11. URL: `https://ant.design/`.

[42] The Chromium Projects. *Chromium - The Chromium Projects*. URL: `https://www.chromium.org/Home`. (Accessed: 2020-12-15).

[43] webpack. *webpack*. URL: `https://webpack.js.org/`. (Accessed: 2020-12-15).

[44] Debra Littlejohn Shinder. *Computer Networking Essentials*. First Edition. Cisco Press, 2001.

[45] Matt Dee. *Inside LAN Sync*. URL: `https://dropbox.tech/infrastructure/inside-lan-sync`. (Accessed: 2020-12-14).