UNIVERSITY OF APPLIED SCIENCES RAPPERSWIL
DEPARTMENT OF COMPUTER SCIENCE

STUDY PROJECT

Redbackup: A Redundant Distributed Backup System Prototype

Authors:
Fabian HAUSER and
Raphael ZIMMERMANN

Advisor:
Prof. Dr. Farhad MEHTA

Autumn Term 2017

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

DON'T PANIC

"It looked insanely complicated, and this was one of the reasons why the snug plastic cover it fitted into had the words DON'T PANIC printed on it in large friendly letters."

The Hitchhiker's Guide to the Galaxy

Abstract

Fabian HAUSER and Raphael ZIMMERMANN

Redbackup: A Redundant Distributed Backup System Prototype

Today, most individuals and small enterprises have a limited choice of how and where they backup their data.

One possibility is via local storage media, for instance using external hard disk drives. This requires considerable administration and may lead to a single point of failure, since location redundancy requires extra effort. A second possibility is to use cloud backup storage. This may lead to issues of privacy and a high dependency on third-party providers.

There are no easy-to-use distributed backup systems with private data storage on the market today.

In this study project we propose a redundant distributed backup system to address this issue.

The architecture of this system consists of backup nodes which exchange data using a peer-to-peer protocol, as well as a client application that creates and restores backups. A management system has been introduced to allow users to manage multiple backup nodes.

As a proof of concept, a prototype of the proposed client and node applications with a reduced feature set has been implemented in the Rust programming language.

Management Summary

Motivation

Today, most individuals and small to medium enterprises make backups on cloud backup storage or local storage media as e.g. hard disk drives or network attached storage systems (NAS).

These solutions require either considerable administrative effort for local storage maintenance or a high level of trust in a third party storage provider.

Currently, there are no backup systems available on the market which are both easy-to-use and provide the user with a high level of data security and privacy.

Project Goals, Approach and Technology

A backup system which resolves these issues must not only provide a secure and reliable application to create and store backups, but also permit users without further domain knowledge to install and configure the application.

To meet these requirements we further analysed and created a comprehensive architectural design.

The subsequent implementation of an architecture prototype took place in the ***Rust*** system programming language[1] which we learned during the course of this project. Rust enabled us to create a very stable yet efficient backup prototype.

Results

The architecture consists of backup nodes which store and distribute data directly over a network connection and a client application that creates and restores backups to or from nodes. Lastly, a management system is introduced to allow users to manage multiple backup nodes.

The presented prototype demonstrates the viability of our proposed architecture, introducing a reduced feature set. The prototype can create, distribute and restore unencrypted backups.

Prospects

To extend the prototype into a fully functional backup system, there are multiple functionalities and improvements that may be implemented. The main missing parts are backup encryption, splitting of backup data, advanced data distribution strategies and the management application.

With our prototype, we demonstrate the viability of the architecture and pave the way for further implementations.

[1]For more information, see `https://www.rust-lang.org/`

Acknowledgements

We would like to thank our advisor, Prof. Dr. Farhad Mehta, for his continuous support and helpful comments.

Furthermore, we would like to thank Andrea Jurt Massey for her feedback regarding writing and language use.

Contents

Chapter 1

Introduction

1.1 Motivation

In this section, we legitimate this study project and explain the value and applicability of our proposed solution.

1.1.1 Present situation

With the ongoing fast digitisation, the demand for reliable and easy-to-use backup solutions is growing fast. Not only enterprises, but also individuals take a great interest in securing their digital artefacts.

Today, most individuals and small enterprises have limited choice with regards to data backup storage.

Cloud backup storage offers many users high quantities of comfortable, easy to use backup storage. In most cases, these solutions post either encrypted or unencrypted copies of the user data into a public cloud environment using specialised software.

Examples for such cloud backup storage providers are Dropbox[1] and Crashplan[2].

Local backup solutions, for instance external hard disk drives or network attached storage systems (NAS) offer a high level of data privacy.

Existing software solutions, for instance Borg Backup [22], rdedup [5] or custom implementations with rsync [10] and ceph [40] allow for safe, deduplicated backups.

An existing solution which is easy to use for the Apple Mac platform, is Apple Time Machine combined with a Apple Time Capsule[3].

1.1.2 Problem

The backup solutions described in the previous section come with several repercussions.

Cloud backup storage requires a high level of trust in a third party provider. It is not evident, what level of data security and availability a user is provided with.

Furthermore, such solutions may raise privacy concerns or even legal issues, as data is not kept safely on premises but in a data centre, possibly in another legal domain.

[1]`https://www.dropbox.com/`
[2]`https://www.crashplan.com/`
[3]`https://www.apple.com/airport-time-capsule/`

Local backup solutions require considerable administrative effort, e.g. by managing and exchanging hard disk drives.

Another problem that is often encountered is missing backup copies. For instance, backups are only stored on a hard disk drive in one location - which might lead to data loss, e.g. in case of fire.

Safe and reliable solutions are non-trivial to set up and require a high level of knowledge to operate.

Secondly, most of these solutions create backups directly to a writeable storage medium from the client computer and do therefore not prevent data loss in case of ransomware infections [43].

1.1.3 Solution

The problems listed in the previous sections must be addressed in the form of a fast, easy to use, and secure backup system. Such a system must combine privacy benefits of local backups with automated data distribution, to provide high safety guarantees.

The target users of such a system are individuals (e.g. families) and small enterprises.

1.2 Goals and Tasks

This section presents the revised goals and provides a rationale for the deviations from the original goals.

1.2.1 Initial Goals

The initial goals of this study project where specified by us in cooperation with Prof. Farhad Mehta in the Task Description.

1.2.2 Revised Goals

This section lists the revised goals that were specified during the beginning of the project. All deviations from the Task Description are noted in the following section 1.2.3 Deviations from the Original Goals.

1. Elaboration of the following issues in a theoretical concept and architecture:
   (a) joining of nodes
   (b) planned and unplanned leaving of nodes
   (c) distribution of data within the network
   (d) uploading data into the distributed system
   (e) addressing within the distributed system
   (f) retrieving stored data
   (g) scalability for up to several 100 nodes where every node can store a data volume of up to 2 terabytes.
2. Evaluation of an appropriate implementation language for the prototype
3. Implementation of a prototype, demonstrating the core features as specified in the concept paper.

1.2.3 Deviations from the Original Goals

Degree of Redundancy

While researching data redundancy strategies, we realised that the full specification and implementation of *client m-replication* is not a feasible goal during the study project. The reason for this is its complexity and the limited time frame of the study project, as discussed in section 2.2 Fundamental Design Decisions.

Simplifications for Prototype

Due to time constrains of the study project, it was not possible to implement the full specified architecture in the prototype. Hence, we decided to implement the core backup, restore and distribution mechanisms, to demonstrate that the concept works.

Simplifications for the prototype are discussed in Chapter 2 Architecture Concept Paper.

1.3 State of the Art

In this section, we describe previous work and existing applications in this area.

1.3.1 Backup Applications

During our research, we primarily focused on software that was either described in academic papers or available under open source licenses.

Borg Backup [22] is the most promising candidate of a deduplicating, encrypting backup system, providing most of the described features.

Borg can also create backups to remote locations (e.g. over SSH) where a server-mode Borg instance is running on the remote location. The downside of this implementation is that the client still needs full write access to the backup server and hence would permit malware to delete backups possibly.

Rdedup [5] is an implementation of a backup software similar to Borg in Rust. It is still in an early development stage and is not yet able to create backups to remote destinations.

Rsync, [10] designed initially for file synchronisation, is also often used to create backups today. By using filesystem-hardlinks, it has file-deduplicating capabilities and can also synchronise files to remote locations, so that old backup cannot be modified.

The two most significant downsides of rsync are that it must be combined with several other applications (e.g. SSH, bash scripts, ceph) to provide a secure backup solution and that it is relatively complicated to set up and configure.

1.3.2 Peer-to-Peer Backup Storage

There exist different approaches to create distributed, encrypted peer-to-peer storage systems. Two notable representatives that create such a system decentralised over the internet are Tahoe-LAFS [21] and IPFS [2].

These systems distribute data over multiple (third) parties, consuming and providing data storage. As such, the data security is based on encryption algorithms only.

Another well-researched market is that of distributed filesystems, with prominent representatives as the Andrew File System [13] and ceph [40]. These filesystems are commonly used in low latency, directly connected environments as data centres.

There also exist multiple concept papers on peer-to-peer distributed backup storages. The reports "Adaptive Redundancy Management for Durable P2P Backup" [7] and "On Scheduling and Redundancy for P2P Backup" [37] examine ways how such a system might distribute data in an efficient manner.

1.3.3 Goals

In this study project, we focus on the architecture of a backup software to lay the foundations for a practical solution used by small enterprises and individuals. The architecture should provide developers with a clear guideline on how to implement such a system. Additionally, the prototype demonstrates a minimal implementation of such a system.

Chapter 2

Architecture Concept Paper

In the first section of this chapter, we propose and discuss a system architecture for a backup system. In the second section, we explain the underlying design decisions. The third section presents the structure and test results of our prototype, which implements a subset of the proposed system.

We illustrate architecture structure using the C4 model for software architecture[1].

2.1 System Architecture

Actors There are two kinds of actors interacting with the system. A typical *user* wants to store backups in the redbackup system and restore them (partially) when needed. The other kind of actor is an *administrator* who configures the system, e.g. extends storage capacity or replaces corrupted disks.

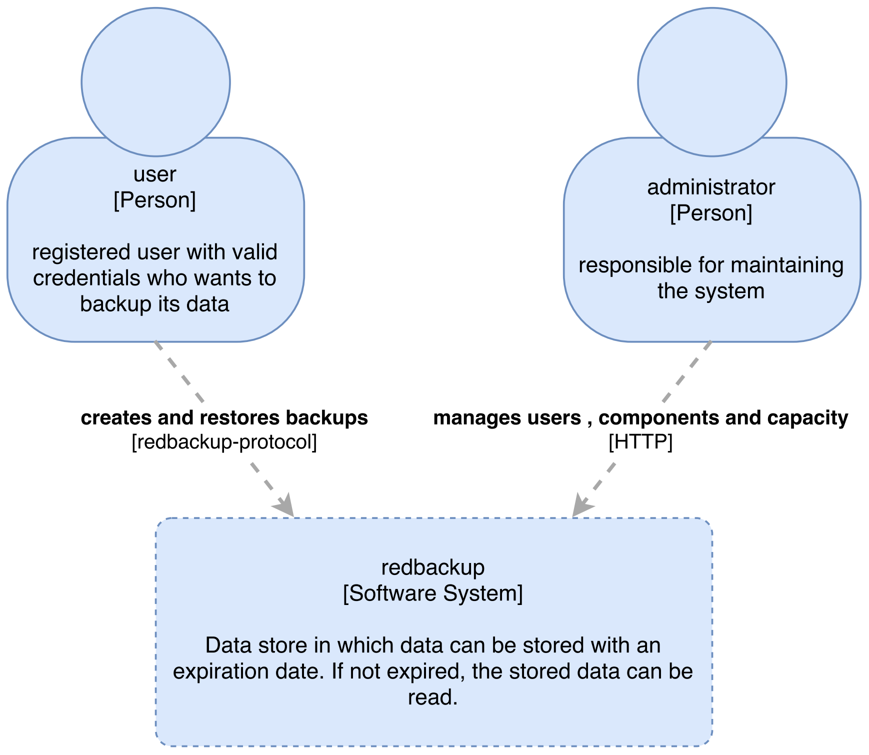
user
[Person]
registered user with valid credentials who wants to backup its data
administrator
[Person]
responsible for maintaining the system
creates and restores backups
[redbackup-protocol]
manages users , components and capacity
[HTTP]
redbackup
[Software System]
Data store in which data can be stored with an expiration date. If not expired, the stored data can be read.

FIGURE 2.1: C4 System Context diagram showing the big picture.

Distinctive of both actors is that they do not want to interact directly with the system unless human interaction is inevitable. This takes the burden of manually

[1]https://c4model.com/

creating backups away from the user including the risk of oblivion and minimises management efforts required by the administrator.

Both actors, as well as their intentions, are described in more detail in Appendix A.4 Architectural Concept Paper. Figure 2.1 presents a high-level overview that illustrates the interactions of the actors with the redbackup system.

Structure The redbackup system consists of four core components as shown in the C4 Container diagram in Figure 2.2.

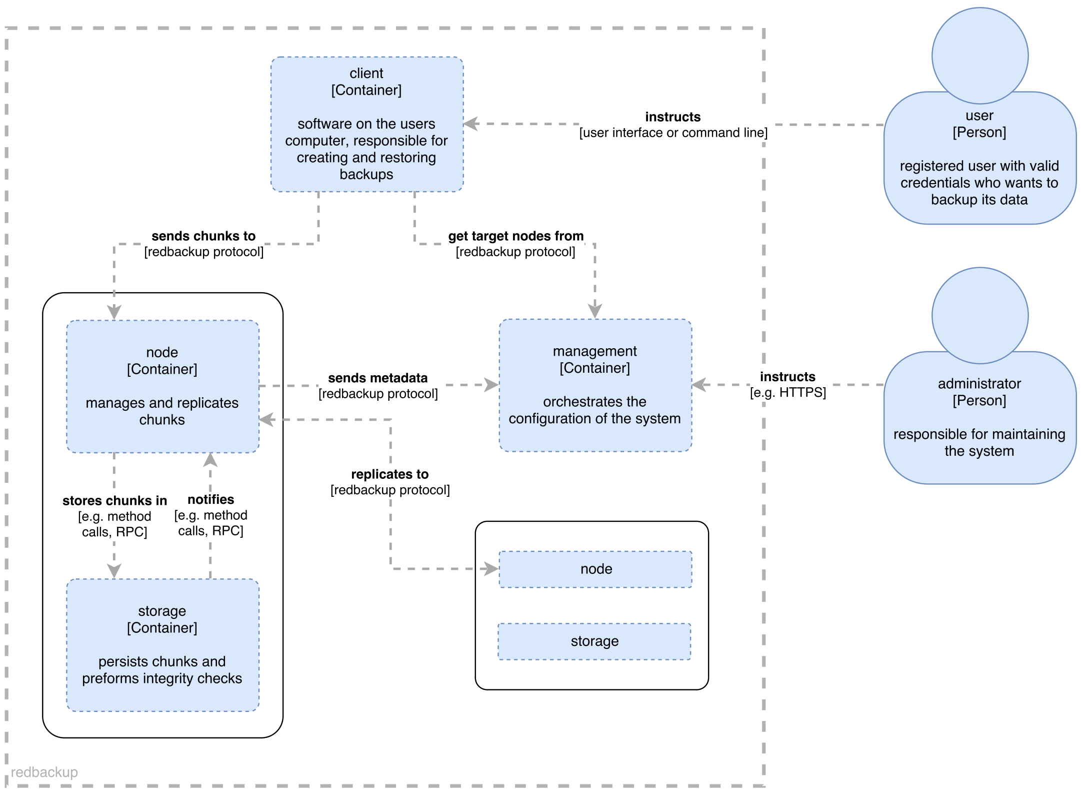

FIGURE 2.2: C4 Container diagram illustrating the high-level shape of the redbackup software system and how responsibilities are distributed.

Client A *user* instructs a *client* program typically running on the users machine to perform (unattended) backups and restores. A *client* persists and loads its data from one or more interconnected *nodes*.

Node A *node* is in charge of data *chunks* including their replication onto other *nodes*. *Nodes* persist the actual data in a separate component, a *storage*, to encapsulate persistence from replication and interaction to support different kinds of storage technologies (e.g. plain file systems or databases). A *node* and its *storage* are typically deployed on the same host.

Management One central *management* component orchestrates the configuration of the system by providing metadata to *clients* and *nodes*. This metadata includes user information and a set of all *nodes* in the system including their addresses and states. *Clients* and *nodes* must cache this metadata which ensures that a temporary unavailability of the *management* component does not compromise the replication and backup process.

All components including their responsibilities and interactions are described in more detail in Appendix A.4 Architectural Concept Paper.

Protocols Redbackup specifies a high-level protocol that is used for internal communication (noted in all C4 diagrams as ***redbackup protocol***). We deliberately specified the communication on a high level to encapsulate the underlying transport mechanisms.

2.1.1 Backup creation

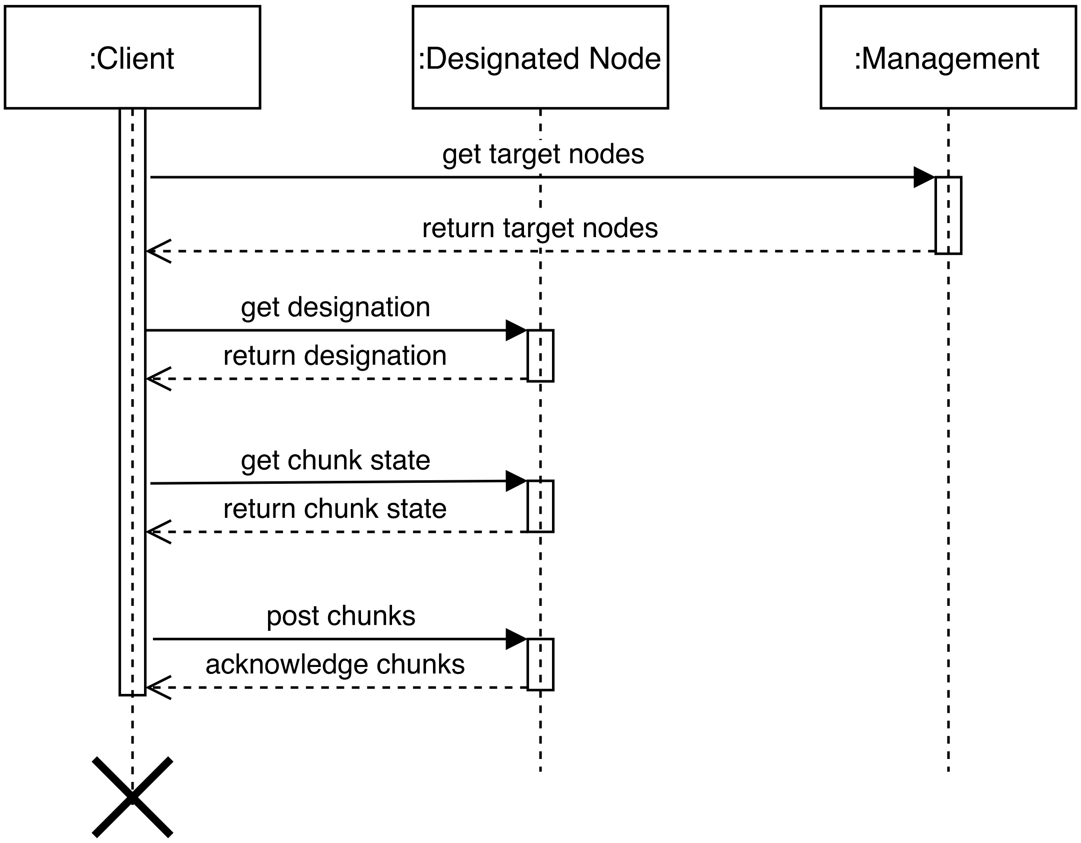

FIGURE 2.3: UML sequence diagram illustrating the communication during the creation of a backup.

To create a backup, a *client* loads all *node*-metadata, that includes the addresses, from the *management*. The *client* caches this set and chooses one of them as *designated node*. This selection can either be random or based on heuristics, e.g. by analysing the round trip time to a *node*.

The *client* then requests permission to perform a backup on a given *designated node* by sending a ***get designation*** message that includes an estimated backup size and the *expiration date*. If the *designated node* has the storage capacity as well as other resources available (i.e. it is not under overload), it confirms the designation request.

By now, the *client* must start to build up backup metadata - hereafter called *chunk index* - by walking recursively through all directories to backup. Each file that is not explicitly excluded is split up into one or more data *chunks* using a rolling hash[23].

Each of these *chunks* are then encrypted individually. Afterwards the *client* derives the identifier of every *chunk* based on its contents. This mechanism enables deduplication. Section 2.2 Fundamental Design Decisions discusses *chunks* and *chunk identifiers* in detail.

The *client* sends the calculated *chunk identifiers* at regular intervals to the *designated node*. The *designated node* returns a subset of all *chunk identifiers* of the *chunks* that are already present on it.

The *client* then posts the missing *chunks* to the *designated node* which acknowledges successful receipt.

When all *chunks* are transmitted successfully, the client encrypts the *chunk index* as well and sends it to the *designated node* using the same mechanism as for any other *chunk*. The only difference is that an additional flag - hereafter called *root handle* - is set so that it can be found again for restore.

The detailed backup scenario can be found in Appendix A.4.4.1 Create Backup.

2.1.2 Backup restore

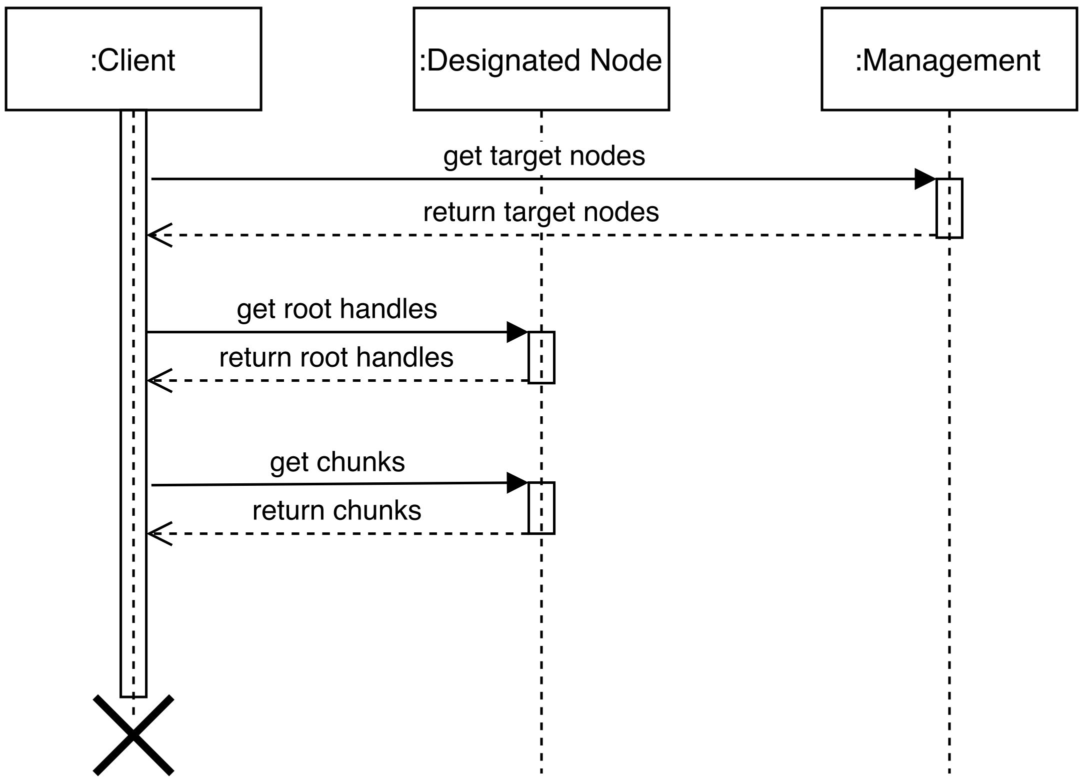

FIGURE 2.4: UML sequence diagram illustrating the communication during the restore of a backup.

The restore mechanism works identical to the backup process up to the point where a *designated node* is chosen.

Next, the *client* loads all *root handles* from the *designated node*. The *client* can decrypt the *chunk tables* from the returned *chunk contents*. Based on this information, the *client* can provide multiple restore options to the *user*, e.g. to restore a particular version of a given file. With the user input and the previously fetched *chunk tables*, the *clients* can calculate which *chunks* must be requested from the *designated node*. *Chunk contents* are then downloaded at regular intervals from the *designated node*, decrypted and combined.

2.1.3 Replication

We deliberately only specified *system n-replication*, which means the degree of redundancy for the entire system is equal to the amount of *nodes* in the system (see 2.2 Fundamental Design Decisions).

A *node* is in charge of all data stored on it. Each *node* - hereafter called *sending node* - randomly picks *n* of its data *chunks*. It then picks one other *node* randomly - hereafter called *designated node* - and requests which of the chosen data *chunks* remain persisted on the *designated node*. The *designated node* returns a subset of the requested *chunks* that consists of all *chunks* that it owns. Using this response, the *sending node* sends all missing *chunks* to the *designated node* which acknowledges successful receipt.

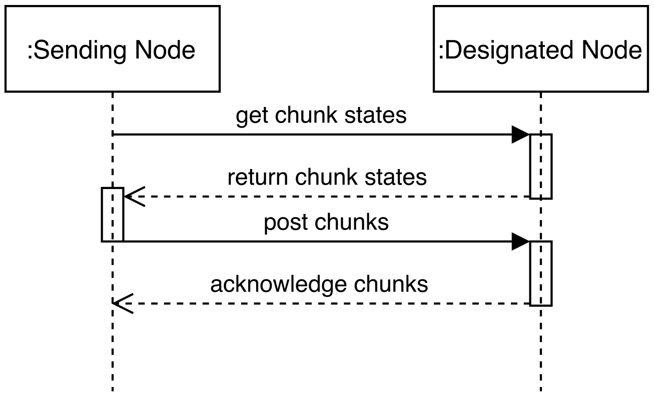

FIGURE 2.5: UML sequence diagram illustrating the communication during replication.

This scenario is described in more detail in the Appendix A.4.4.6 Data Replication.

2.1.4 Security and Encryption

Data Integrity One of our primary design goals is to ensure that once data is in the redbackup system, it cannot be altered or deleted from a *client* to prevent ransomware attacks [43]. To achieve this, each backup is provided with an *expiration date* on which *nodes* are allowed to remove the associated data.

Reasoning and potential risks of physical time are discussed in section 2.2 Removal of Old Backups.

Data Integrity on Nodes Because *nodes* can also be the target of randsomware attacks, each *node*, or more precisely its associated *storage* component, must verify that given *chunk contents* are not corrupted. To be able to do so, it must be possible to calculate the identifier of such a data *chunk* from its contents, using cryptographic hash functions.

A discussion on the role of hash functions including the chosen algorithms for the study project is carried out in section 2.2.1 Hash Collisions.

Data Integrity on the Management The *management* component has no knowledge of the persisted data in the system. It only manages configuration and must not have detailed knowledge of *chunks* (need-to-know principle [20]).

Data Encryption A *client* encrypts *chunks* when creating a new backup. This ensures that no other participant in the redbackup system can inspect file contents (need-to-know principle [20]). The encryption and decryption keys are only stored on the *client* and must be backed up separately.

Transport Security All sent *messages* should be signed by the sender. If so, transport layer encryption is not strictly necessary because all user data is already encrypted on the *client* with the only exception of the *expiration date*.

A detailed description of these security mechanisms would be out of scope for this study project and will have to be carried out in a next step.

2.1.5 Partitioning & Scaling

Availability & Overload To scale the redbackup system regarding availability, more *nodes* can be added to the redbackup system. If a given *node* is overloaded, a *client* can use another *node* for backup creation or restore.

Because the backup and restore processes require approval of a *designated node* (see scenarios A.4.4.1 Create Backup and A.4.4.2 Backup Restore), an overloaded *node* can finish work in progress backups/restores and reject new requests (following the patterns Finish Work In Progress and Shed Load [11]). The data is replicated to overloaded *nodes* eventually.

We ensured that *nodes* are (mostly) stateless which simplifies scaling as well.

Storage Scalability Because the proposed system currently only supports *system n-replication* (see section 2.2 Fundamental Design Decisions), the maximum storage capacity is equal to that of the *node* with the smallest storage. Therefore, to increase storage capacity, the capacity of all *nodes* must be extended to meet the desired amount.

To minimise network usage deduplication of data is used on the *client*. Deduplication is discussed in the paragraph Storage Unit in 2.2 Fundamental Design Decisions.

2.1.6 Failure Detection

Additionally to the above-discussed mechanisms for fault tolerance regarding security and scaling, the following failure detection mechanisms are in place.

Reporting Most issues that occur on a *node* are reported to the *management* component (following the pattern Someone in Charge [11]). The *management* can then decide to notify the system administrator or execute error mitigation processes, e.g. suspend a given *node* temporarily.

If a *node* tries to connect to another *node* that is not available, it notifies the *management* component (following the pattern System Monitor[11]).

Each *node* periodically checks the persisted contents for possible corruption using the checksum mechanisms discussed in 2.1.4 Security and Encryption (following the Pattern Routine Audits [11]). If a corruption is detected, the *storage* notifies the *node* which notifies the *management*.

Single Point of Failure For the case that the *management* component is temporarily unavailable, *nodes* and *clients* cache metadata, e.g. information about other *nodes*. Using these caches, *nodes* can perform replication and *clients* manage their backups without interruption. Any notifications that were not successfully transmitted to the *management* must be buffered on the *nodes* in order to ensure their delivery.

2.2 Fundamental Design Decisions

We used the morphological box technique to explore different possible implementation options (see Table 2.1). The chosen option should be as simple as possible for the prototype developed in the study project but extensible for further adaption.

The following paragraphs reason the selected entry in each dimension.

Redundancy We originally planned to support *client m-replication*, which means that the *client* defines a custom degree of redundancy from 1 to the number of *nodes* in the system. However, this is a complex mechanism that requires sophisticated algorithms to work correctly and efficiently. For the prototype, we chose the more straightforward implementation option system *system n-replication*, where the degree of redundancy for the entire system is equal to the number of *nodes* in it. Changing this option in the future will not be trivial because it requires additional changes in the replication process and the communication protocols.

Storage Unit The idea of ***chunks*** come from Borg Backup. Files are partitioned into chunks using a rolling hash which enables deduplication and space efficient backups for large files [23]. These are desired properties in a backup system to minimise network and disk usage.

Encrypting chunks means that deduplication of the same file coming from different users is not possible anymore but is a necessity for privacy. Encryption is not trivial and requires a user concept that is out of scope of the prototype developed in the study project. We chose the ***plain files*** option for the study project to simplify the *client* implementation. In the future, supporting ***encrypted chunks*** will be possible by just modifying the *client*.

Role of the Management The ***one in charge*** option is the most straightforward option to implement, but conflicts with many intentions of the administrator (see A.3.2 Intentions of an Administrator). We also intended to avoid a single point of failure. We chose the option ***autonomous replication*** because it guarantees that replication is always ensured and keeps communication relatively simple.

Storage Backend Using the file system is the simplest possible solution for the study project and therefore the selected option. Adding support for other backends in the future is still possible because the storage component is an isolated part in the architecture (see A.4.3.3 Storage).

The number of files in a folder is limited depending on the used file system, length of a filename and other factors. Some file systems (e.g. ext4) have a global limit for the maximal number of files. This limit is 4 billion files for ext4. [42]. Therefore we use the ext4 file system to persist data in the study project.

Removal of Old Backups We decided to use ***physical time***stamps that must be specified on backup creation. After expiration, the backup data may be removed by a garbage collector running on a *node*. This may be extended to allow only mutual garbage removal in the future.

A significant problem that ***physical time*** addresses is the safety of backup data in case a user computer is infected with malware [43]. An illicit application might command the removal of backups or create new backups to initiate a garbage collection process to free storage capacity.

Nevertheless, the use of ***physical time*** has the downside of possible data loss in case of wrong system times. To mitigate this risk, the system should use multiple distinct upstream time-servers. Multiple distinct upstream time-servers are used with a high probability as the proposed redundancy model motivates users to expand the system across multiple physical locations. Furthermore, the *client*, *nodes* and *management* should verify a reasonable accurate time when communicating mutually.

Programming Language & Ecosystem We chose Rust over Erlang and Go because it offers great performance and minimal overhead supported by a powerful type system. A complete language evaluation can be found in A.5 Language Evaluation.

2.2.1 Hash Collisions

To achieve deduplication and space-efficient backups for large files, as discussed above, a chunk identifier must be derived from the actual chunk contents. A common mechanism used to derive identifiers from binary data is the use of cryptographic hash functions. Most cryptographic hash functions produce a message digest that has a fixed length (e.g. SHA-256[9] produces a 256-bit digest) for a given message with an arbitrary length. The restricted length can theoretically lead to collisions. Perfect hash functions do not have this property because their input message length is equal to the length of the resulting message digest. A perfect hash function is not practical in our case due to the large message digests. With cryptographic hash functions, collisions are possible but unlikely. Assuming that the applied function does produce equally distributed results, the probability can be calculated based on the birthday problem[41] as follows, where p is the number of chunks in the system and n the length of the message digests:

$$P(p,n) = \frac{p^2}{2^{n+1}}$$

Assuming we have $p = 30^{21}$ chunks the system (which is equivalent to two billion years of music assuming each chunk has a size of one byte[19]) and using the SHA-256 algorithm, the probability of a collision is about $4.72 \cdot 10^{-16}$, which is highly improbable and may therefore be neglected.

However, did a collision occur after all, for example, if the used cryptographic hash function were flawed or the unlikely event occurred, it would result in data loss.

In theory, we could detect collisions on the *client*. To do so, every time an identifier is calculated, the *client* must verify that if a chunk with the same identifier already exists in the system, it has the exact same contents. If the contents differ, it is a collision. This approach requires a lot of network traffic and can slow down the backup process significantly.

TABLE 2.1: Morphological Box

Redundancy	No redundancy	*Client m-replication*: The *client* defines a custom degree of redundancy (from 1 to the number of *nodes*).	*System m-replication*: The administrator defines the degree of redundancy for the entire system (from 1 to the number of *nodes*).	***System n-replication***: The degree of redundancy for the entire system is equal to the amount of *nodes* in the system.
Storage unit	**Plain files**	Encrypted files	Chunks: Cut files into multiple parts and store these individually.	Encrypted chunks: Same as chunks, but every chunk is individually encrypted.
Role of the management	One in charge: The management knows and controls everything (e.g. the location of every file/chunk).	Configuration only: The management must be available for administrative tasks only. The *nodes* are mostly autonomous.	**Autonomous replication**: The management must be available for most of the tasks but replication also works if the management is down.	No management: Every *node* is completely autonomous.
Storage backend	**Plain filesystem**: Store all files/chunks as files in one directory with a unique identifier.	Database: Use an existing database solution (e.g. Git, Redis, RocksDB).	Cloud Storage: A proxy to a cloud storage provider (e.g. Amazon S3).	Custom: An optimized version of the plain file system option with optimised indexing and compression.
Removal of old backups	**Physical time**: Data is removed on a specified physical time.	User command: The user commands removal of data.	Free storage: Data is removed, as soon as capacity issues occur.	Physical time with mutual agreement: All *nodes* must agree before data is removed.
Programming language / ecosystem	**Rust**	Go	Erlang	

Another place to detect collisions is on the *node* component. A *node* can verify if the contents of a given chunk are equal to the contents already present in the system. The downside of this approach is that it requires the *client* always to send the full contents of every chunk, which leads to a lot of additional network traffic.

Both of the described approaches for collision detection have significant costs that are not practical.

As for the study project, we use the SHA-256 algorithm[9] and neglect the risk of hash collisions due to its low probability. Nonetheless, we prepare all protocols and components to use an interchangeable mechanism for the calculation and transmission of file/chunk identifiers.

2.3 Prototype

Because the entire system, as described in the previous sections, is too large and complex to implement in the form of a study project prototype, we reduced the functionality to its core.

Our prototype focuses on the backup, restore and replication scenarios as described in Appendix A.4.4 Scenarios leaving out encryption.

We also limited the supported platforms to 64-Linux only as this is the operating system we use for development and continuous integration.

The implemented prototype is organised in 46 modules, defines 188 functions and has 4’240 lines of code, including unit test code and whitespace. All components are implemented using the Rust programming language [34]. Installation instructions are documented in the project’s repository.

2.3.1 Concrete Architecture

Figure 2.6 illustrates the system as implemented in our prototype.

The *client* and *node* are delivered as executables that are configured and launched using the command line.

The *node* component binds itself to a configured network interface and port on which it provides the services for backup creation, backup restore and replication.

The *client* executable is started individually for every operation that is the creation of a backup, listing all backups persisted on a given *node* and data restore.

Client

The *client* executable bundles three components as shown in Figure 2.7.

Client-cli contains the command line specific logic that provides an uncluttered interface for advanced users and serves as an entry point in the client’s core logic. We used the clap[2] library to implement this component. The command line interface is described in Appendix A.6.

The ***client*** component contains the actual logic for creating and restoring backups. It is organised as a library so that it can be used from other projects as well, e.g. if we provide a graphical user interface in the future. The *client* component creates the *chunk index* for every backup in a separate ***SQlite-database***. For database access, we used the Diesel[3] ORM-library.

[2]https://clap.rs/

[3]https://diesel.rs/

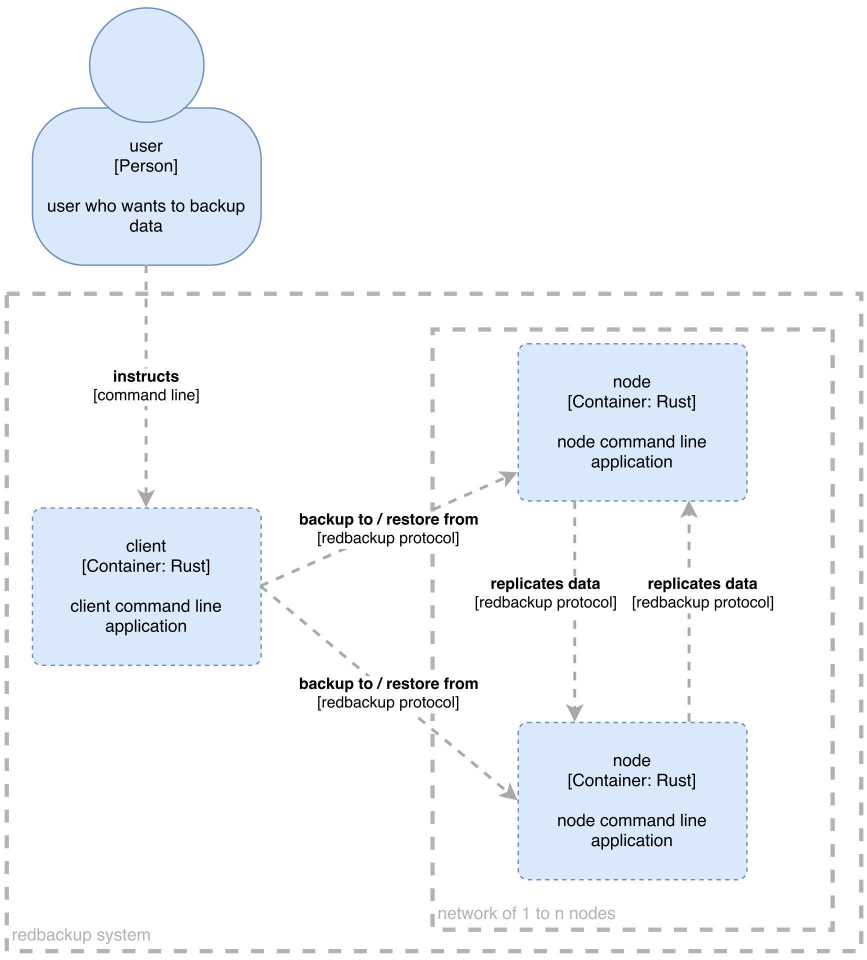

FIGURE 2.6: C4 Container diagram illustrating the high-level shape of the prototype and how responsibilities are distributed as implemented in the study project.

The *client* component makes heavy use of a networking library called tokio[38] that provides an efficient event loop similar to the Reactor pattern [4]. Although tokio supports highly parallel networking code, we decided to implement all interactions on the *client* serial to maintain readability.

The (de-) serialisation mechanisms for messages sent to and received from *nodes* are encapsulated in the ***protocol*** component, following the Forwarder Receiver Pattern [4].

We opted for Diesel and tokio because there are currently no comparable alternatives on the marked in both application areas.

In the prototype, the *client* interacts with one *node* at a time to maintain simplicity. The *node* to interact with is passed to the *client* executable via command line arguments.

Node

The *node* executable bundles four components as shown in Figure 2.8.

Just like the *client* implementation, the command line logic is encapsulated in a separate component called ***node-cli***.

The core logic is implemented in the ***node*** library. Like the *client*, the *node* component makes heavy use of the tokio and Diesel libraries. In contrast to the client,

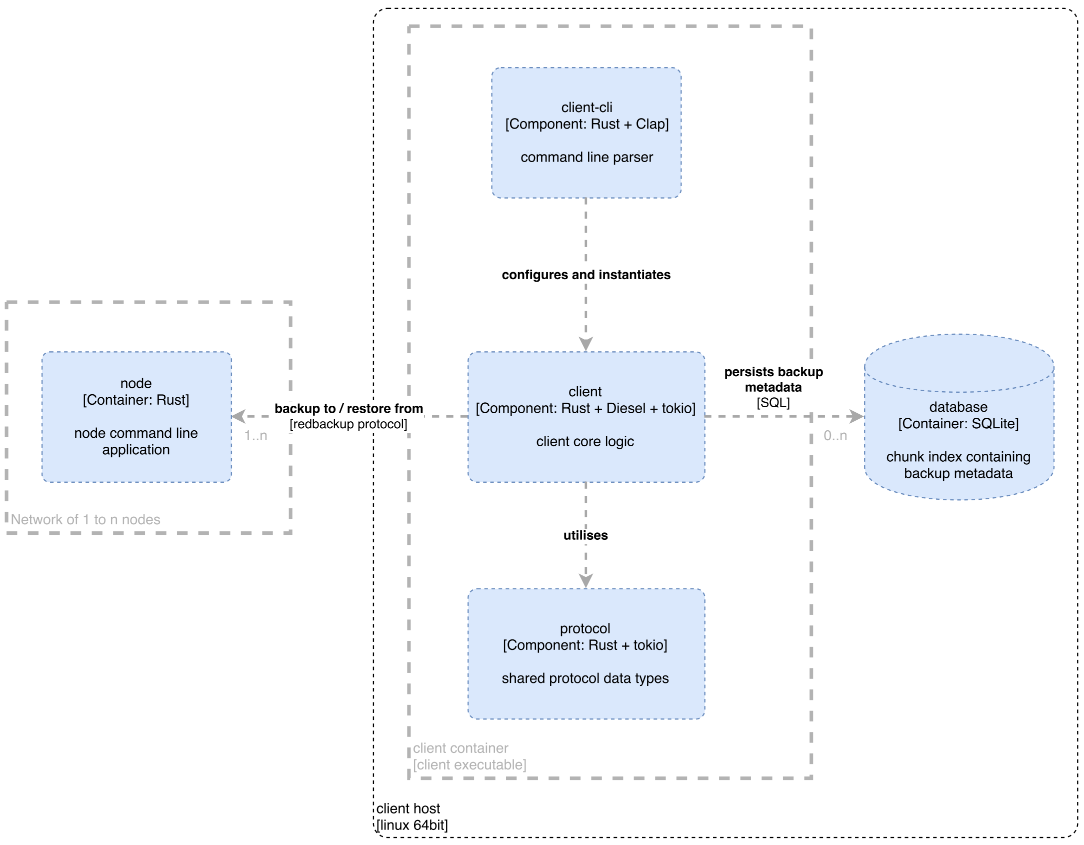

FIGURE 2.7: C4 Container diagram illustrating the shape of the *client* and how responsibilities are distributed as implemented in the study project.

we used the parallel features of tokio. To keep the chosen technology close to the client, we decided to use SQLite for the *chunk table*. This must be changed in the future because SQLite locks the entire database when writing, which makes concurrent updates impossible [36].

We use the same ***protocol*** component for (de-) serialisation of messages on the *node* and the *client*.

The ***storage*** component is also bundled directly in the *node* executable. It persists data in one single directory as described in 2.2 Fundamental Design Decisions.

Other known *nodes* are passed to the *node* executable via command line arguments.

Because user authentication requires additional cryptographic efforts, the prototype accepts backups and replications from everyone.

2.3.2 Testing

In the following subsections, we describe how the prototype and architecture can be tested.

Unit Tests

Test Driven Development (TDD) [1] should be used as much as possible. Our Definition of Done (Appendix A.2) states that ***reasonable unit and integration tests [must] exist and pass.***

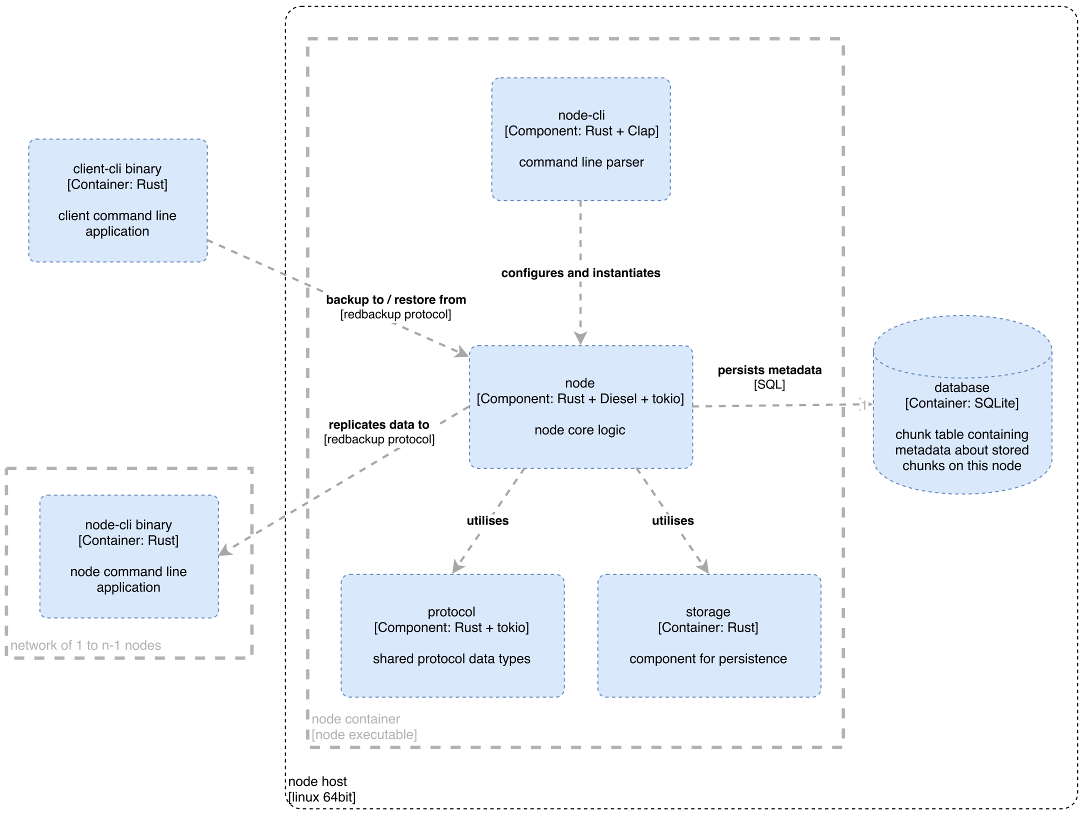

FIGURE 2.8: C4 Container diagram illustrating the shape of the *node* and how responsibilities are distributed as implemented in the study project.

All unit tests are executed on every build run on our continuous integration server. That is on every repository push and pull request.

Integration Tests

We defined two primary environments for the integration tests: A minimal network as defined in Figure 2.9 and a medium one, as defined in Figure 2.10.

These two rather small network styles will most commonly be deployed, yet they can expose most of the possible problems.

The integration tests are run automatically at least on every tagged release (i.e. at least once every sprint). Because of the (yet) well manageable set of integration tests for the prototype, these tests can run on every build as well.

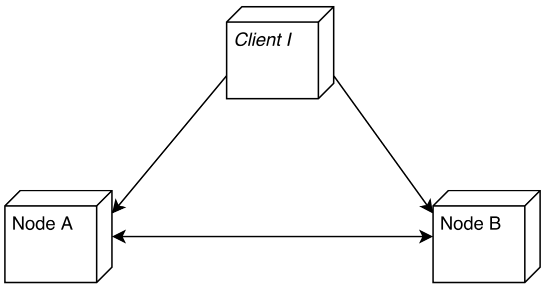

FIGURE 2.9: Minimal integration test with one *client* and two *nodes*.

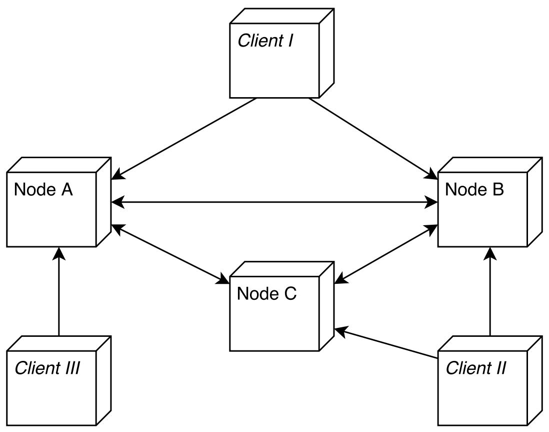

FIGURE 2.10: Medium integration test with three *clients* and three *nodes*.

Tests for fault tolerance, e.g. what happens if a *node* goes down while receiving data, can be implemented as integration tests as well.

Architecture tests

Architectural tests are special and manually run tests to verify the scalability of our software architecture.

Chapter 3

Discussion and Conclusion

This chapter contains our achieved results and lessons learned, discusses future work and closes with a conclusion.

3.1 Achieved Result

3.1.1 Prototype

In our prototype, we implemented the basic concepts of our architecture in a limited form. Rust was undoubtedly an excellent choice as a programming language for the prototype regarding robustness and performance but turned out to have a very steep learning curve. The same applies to the used frameworks tokio and Diesel. For this reason, we had to compromise to demonstrate as much of the architecture as possible without spending too much time on learning Rust.

Nevertheless, our implementation is solid and proves that the proposed architecture is robust and can be pursued further.

3.1.2 Prototype Test Results

Unit Tests We used test driven development (TDD) to develop the prototype as much as possible, which turned out to work great in the Rust ecosystem.

Some unit tests written for the prototype are not pure unit tests but minimal integration tests. Because Rust is not a traditional object-oriented language, it is not possible to introduce and use interfaces (Traits) in the same way as we were used to from other languages such as Java or C#. Due to the steep learning curve of Rust, we were not able to fully utilise the corresponding mechanisms.

Integration Tests To write comprehensive black box integration tests, we implemented a testing library written in Python[1]. In this framework, the internals on how to launch and configure *clients* and *nodes* is encapsulated in classes. Using this abstraction, we were able to launch *clients* and *nodes* in separate Docker[2] containers, so that they are as isolated as possible. All containers used in a test case are connected to a dedicated Docker network, which eliminates possible interferences with other network services.

We wrote integration tests that verify that backups are flawlessly created, restored and replicated onto other *nodes*.

[1]`https://www.python.org/`
[2]`https://www.docker.com/`

Test coverage Because Rust is still a young language with a relatively small ecosystem, tools for measuring code quality are still rare and immature. For our unit tests, we used Tarpaulin[3] to generate code coverage. Tarpaulin does not yet cover all language features and therefore returns an incomplete coverage number. We achieved 53.5% line coverage. This number would be significantly higher if all executed lines were counted correctly (e.g. generated code using macros as well as compiler optimisations are not counted).

Code coverage achieved using the integration tests is not yet supported by any tool known to us and therefore undocumented. The integration tests do however cover all positive scenarios that were implemented.

Our integration testing framework allowed us to write such tests in a simple fashion.

3.1.3 Architecture

The architecture we elaborated in Chapter 2 and Appendix A.4 turned out to to be stable. It has proven advantageous that we did not specify too many details at the beginning (for example, the protocol) but focused on the high-level view.

3.1.4 Architecture Test Results

To ensure the validity of the proposed architecture, we manually ran architecture tests. We focused on scalability, data capacity and concurrent backups. We used our integration test framework for these tests as well.

Overall Performance

The conducted architecture tests on the prototype have shown a solid overall performance. Unfortunately, we observed significant memory consumption and CPU utilisation during most test scenarios.

CPU Utilisation Replication and the creation of a backup require a lot of CPU time on the *node* and *client* components. The *client* component must execute many hash functions during the creation of *chunks*. A *node* must verify that the provided *chunk identifiers* can be derived from the sent *chunk contents* during backup creation and replication.

Intensive CPU utilisation can be problematic, especially on a *node*. This is somewhat an inherit problem of the proposed architecture because these calculation are required to ensure the integrity of the data stored in the system.

To mitigate this issue, a queuing mechanism could be implemented on the *node* component that temporarily accepts *chunks* without performing integrity checks. These integrity checks could then be performed in the near future and the sent *chunks* would afterwards be added definitely to the chunk table. The backup and replication protocol needs to be adapted to signal such a temporary queuing on a *node* to *clients* or other *nodes*. A new state (e.g. queuing) could be sent instead of an acknowledgement that instructs *clients* and *nodes* to ask again for acknowledgement later.

It has to be investigated whether specific CPU acceleration for hash calculation could mitigate this problem as well.

[3] https://github.com/xd009642/tarpaulin

Memory Consumption To prevent premature optimisation, which Donald Knuth famously pointed out is the root of all evil [15], we implemented *messages* without streaming support. We implemented the high-level redbackup protocol as framed Message Pack[4] encoded *messages* based directly on TCP. Such a Message Pack *message* is (de-)serialised in a single piece. In other words, whenever a *message* is created or received, all its contents including the payload is loaded into memory. This decision, in combination with the design decision to not split up large files into multiple *chunks*, has led to significant memory consumption.

The protocol details must be clarified in the future to allow *message* streaming. Refactoring the protocol component to use streaming mechanisms is feasible since tokio provides these mechanisms[39].

Size Scalability Test Results

As per our requirements in Appendix A.3, the architecture should scale up to 100 *nodes*.

To test this scenario, we used the same underlying techniques as in our integration tests, but scaled the infrastructure up to 100 *nodes*.

Due to the high memory consumption, we were not able to conduct this test with a significant amount of data. A test run during which a 5MB file was replicated to 99 *nodes* did not indicate a degraded performance.

Data Capacity Test Results

Our requirements (Appendix A.3) also state, that a *node* must be able to handle up to 2TB of data. To test this requirement, we planned to create large amounts of random data that has to be stored. This is a realistic requirement, as e.g. images, audio files and film collections might reach such sizes in practice.

It was not possible yet to create one single backup of 2TB at once due to the high memory consumption. Performing multiple backups in a row of a smaller dataset (i.e. 5 files with a size of 500MB) has not shown a decrease in performance.

Concurrent Test Results

We ran a test in which 5 *clients* backup randomly generated data onto three randomly chosen *nodes*. On average, the entire backup process of a 1MB *chunk* took 90-130ms from one docker container into another. These results clearly support our proposed architecture.

In reality, where *clients* and *nodes* are on separate physical machines, this time will be significantly higher due to network latency. Also, because the creation of backups require a lot of CPU time, running all *clients* on one machine is somewhat problematic. It is likely, that running all *clients* and *nodes* on separate machines would improve the performance slightly.

3.1.5 Requirements and Intentions

Because we aim for fully automated backups with ideally no user interaction at all, we specified the intentions of actors instead of their interactions with the system. Many of these intentions only describe the actors expectations of the system and are

[4] `https://msgpack.org/`

therefore not precisely measurable. We therefore only derived a limited set of measurable requirements from the intentions. Both, the requirements and the intentions, were instrumental during the design phase to make several architecture decisions.

The intentions and requirements are listed in Appendix A.3 Requirements.

We were able to address all specified intentions and requirements in our architecture and prototype.

3.2 Lessons Learned

In this section, we describe unexpected project events and the lessons we learned from them.

3.2.1 Project course

Documentation

While discussing the documentation efforts in mini-retrospective two, we noted that some terms like metadata or chunks were not defined unambiguously and therefore used for different concepts in varying contexts. To standardise these, we decided to introduce a glossary that uniquely and precisely defines each of these terms.

While elaborating the architecture, we started researching advanced data distribution mechanisms and consensus algorithms. We were both very interested in these topics, but after a discussion with Prof. Mehta and retrospective one, we realised that the time frame of the study project would not suffice to implement such advanced algorithms.

We frequently underestimated the documentation efforts, particularly the time required for reviewing. We responded by estimating more time and increase the risk reserve time for documentation issues. Besides, we also agreed we would stop and reassess earlier on issues that took longer than expected.

Rust Formatting and Documentation

During retrospective two, we noted that the source code was not fully formatted according to the Rust Style Guide[5] and that the source code documentation was not complete. To ensure consistent code formatting, we added the RustFmt[6] tool as acceptance criterion to our Definition of Done (Appendix A.2) and created a task to complete the documentation.

Project management

During mini-retrospective one and the first full retrospective, we discussed several small improvements regarding the task management and how, respectively, where we would work together. During the second sprint, we also neglected to plan time for the supervision meeting and infrastructure updates, which we met by creating a checklist for sprint planning.

A month into the project during the second mini-retrospective, we agreed that we should create more issues with shorter running times and make sure that we review issues as soon as possible. Also, the reported working hours were incomplete and

[5]https://github.com/rust-lang-nursery/fmt-rfcs/blob/master/guide/guide.md
[6]https://github.com/rust-lang-nursery/rustfmt#rustfmt—

only narrowly fulfilled the planned sprint goals. Therefore, we decided to log the working hours more precisely and intensify the work efforts.

3.2.2 Decisions

Redundancy: *system n-replication*

For the prototype, we decided to implement *system n-replication*. This decision worked out as we expected and allowed us to create a straightforward yet efficient way to replicate *chunks*.

Programming Language and Ecosystem

During the language evaluation, we decided for using Rust to implement the prototype (See A.5 Language Evaluation for details on this decision).

While we still think that Rust is the right choice for the implementation of a backup application as presented in this report, we would have been more productive with a language we already had experience in, like Python or Java. For a prototype, these languages would also have sufficed, despite possibly not being as stable and fast as a Rust implementation.

Frameworks: Tokio and Diesel

As discussed in Chapter 2 Architecture Concept Paper, we utilised the tokio and Diesel frameworks. While offering an advanced feature set considered the relative young Rust ecosystem, we found that the documentation for both frameworks were not sufficiently comprehensible.

Also, the Diesel framework offers an insufficient set of type implementations for SQLite and lacks extensibility e.g. adding support for timezone timestamps.

Storage: Database with SQLite

As we started implementing the prototype, using SQLite seemed an obvious choice, as it is both easy to use and lightweight.

This decision turned out to be suboptimal, as SQLite is not very well suited for concurrent write access [36] and offers an insufficient set of data types [35]. For example, SQLite only allows signed 32-bit integers to be used as record identifiers, which effectively limits the number of *files*, folders or *chunks* to $2^{31} - 1$ each in the prototype.

As a result of the combined difficulties with Diesel and SQLite, we spent considerably more time implementing the database access than initially planned.

In hindsight, we should have further evaluated other database systems including an in-memory database for the *client*.

3.3 Future work

In this study project, we laid out the fundamental architecture and created a minimal prototype to demonstrate the viability of the main parts of our architecture. On this basis, various aspects can be evolved and improved.

3.3.1 Reduce Memory and CPU consumption

As already stated in section 3.1.4 Overall Performance, the memory consumption and CPU usage can be improved.

3.3.2 Further demonstrate the architecture

As discussed in section 2.3 Prototype, we did not yet implement all functionality as described in the architecture concept paper. Some crucial aspects that we left out still have to be demonstrated, especially joining and leaving of nodes as well as chunk encryption and splitting.

3.3.3 Client-m-replication

As carried out in chapter 2, the details of *client m-replication* are unresolved and have to be carried out.

3.3.4 Evolve the Prototype into a Working Product

If all remaining aspects of the architecture have been demonstrated, an actual working product shall be implemented that is not only a proof of concept but enables users to create backups in a simple, sustainable way.

3.4 Conclusion

In comparison to existing backup solutions presented in section 1.3 State of the Art, we designed a system that is both scalable yet easy to use. Our backup *client* is designed similar to Borg [22] but is solely aimed at backups with the redbackup system, whereas Borg is usually used to create local backups.

We decided against specifying and implementing *client m-replication* in detail, as there is existing research in this area [7] [37] and the time frame of the Study Project would not have sufficed to go into further detail.

Our proposed Architecture has turned out to be accurate as demonstrated with the prototype, in which we implemented the main parts required to perform, restore and replicate backups. We are confident that our design also works on a large scale and can be used to implemented an enhanced backup system for production usage.

Rust turned out to be an excellent choice for implementing a backup software due to its stability, speed and modern language features. Nevertheless, the very steep learning curve resulted in more learning efforts than anticipated.

The Study Project went well from our point of view. Not only were we able to reach most of our ambitious goals, but we also had the opportunity to learn a lot during the project. Our initial planning and the Project Plan turned out to be mostly accurate.

In the Future, we intend to implement a full backup system based on the architecture and prototype elaborated in this study project. The initial vision of an easy to use distributed backup system with private data storage has not only turned out to be realistic but has also been positively received and led to stimulating discussions.

Bibliography

[1] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[2] J. Benet. IPFS - content addressed, versioned, P2P file system. *CoRR*, abs/1407.3561, 2014.

[3] Bundesamt für Sicherheit in der Informationstechnik. Kryptographische Verfahren: Empfehlungen und Schlüssellängen - version: 2017-01. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.pdf?__blob=publicationFile&v=4, 2017.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996.

[5] Dawid Ciarkiewicz et al. Data deduplication engine, supporting optional compression and public key encryption. https://github.com/dpc/rdedup, 2017.

[6] B. Degenhardt. What libraries can I use to build a GUI with Erlang? https://stackoverflow.com/questions/97508/what-libraries-can-i-use-to-build-a-gui-with-erlang/, 2008.

[7] M. Dell'Amico, P. Michiardi, L. Toka, and P. Cataldi. Adaptive redundancy management for durable P2P backup. *Computer Networks*, 83(Supplement C):136 – 148, 2015.

[8] C. Doxsey. *An introduction to programming in Go*, chapter Concurrency. CreateSpace Independent Publishing Platform, 2012.

[9] Eastlake, Hansen, et al. RFC 6234 - shas, hmac-shas, and hkdf. https://tools.ietf.org/html/rfc6234, 2011.

[10] W. et al. rsync - an open source utility that provides fast incremental file transfer. https://rsync.samba.org/, 2017.

[11] R. Hanmer. *Patterns for Fault Tolerant Software*. Wiley Publishing, 2007.

[12] F. Hebert. *Learn You Some Erlang for Great Good!: A Beginner's Guide*, chapter Introduction. No Starch Press, San Francisco, CA, USA, 2013.

[13] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, Feb. 1988.

[14] Software engineering – Product quality – Part 1: Quality model. Standard, International Organization for Standardization, June 2001.

[15] D. E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, Dec. 1974.

[16] N. Matsakis. Introducing MIR - the rust programming language blog. https://blog.rust-lang.org/2016/04/19/MIR.html, 2016.

[17] N. Matsakis and A. Turon. Stability as a Deliverable - the rust programming language blog. https://blog.rust-lang.org/2014/10/30/Stability.html, 2014.

[18] J. Morrow. Packaging Erlang Applications. https://speakerdeck.com/jaredmorrow/packaging-erlang-applications, 2013.

[19] Peter Frisch. 2016: The Year of the Zettabyte - seagate smart business. https://web.archive.org/web/20141208222542/http://storageeffect.media.seagate.com/2014/07/storage-effect/2016-the-year-of-the-zettabyte/, 2014.

[20] M. Schumacher, E. Fernandez, D. Hybertson, and F. Buschmann. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, 2005.

[21] M. Selimi and F. Freitag. Tahoe-lafs distributed storage service in community network clouds. In *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*, pages 17–24, Dec 2014.

[22] The Borg Backup Authors. Borg Backup - a deduplicating backup program. http://borgbackup.readthedocs.io/, 2017.

[23] The Borg Backup Authors. Borg Backup - data structures and file formats. http://borgbackup.readthedocs.io/en/stable/internals/data-structures.html, 2017.

[24] The Erlang Authors. Erlang Programming Language. http://www.erlang.org/, 2017.

[25] The Erlang Authors. Implementation and Ports of Erlang - frequently asked questions about erlang. http://erlang.org/faq/implementations.html, 2017.

[26] The Erlang Authors. Native Implemented Functions - erlang user's guide. http://erlang.org/doc/tutorial/nif.html, 2017.

[27] The Go Authors. Command cgo. https://golang.org/cmd/cgo/, 2017.

[28] The Go Authors. Go Release Cycle for the go programming language. https://github.com/golang/go/wiki/Go-Release-Cycle, 2017.

[29] The Go Authors. Minimum Requirements for the go programming language. https://github.com/golang/go/wiki/MinimumRequirements, 2017.

[30] The Go Authors. The Go Programming Language. https://golang.org/, 2017.

[31] The Go Authors. Why is my trivial program such a large binary? - frequently asked questions. https://golang.org/doc/faq, 2017.

[32] The Rust Authors. Fearless Concurrency - the rust programming language, second edition (draft). https://doc.rust-lang.org/book/second-edition/ch16-00-concurrency.html, 2017.

[33] The Rust Authors. Foreign Function Interface - the rust programming language, second edition (draft). `https://doc.rust-lang.org/book/first-edition/ffi.html`, 2017.

[34] The Rust Authors. The Rust Programming Language. `https://www.rust-lang.org/en-US/`, 2017.

[35] The SQLite Authors. Datatypes In SQLite Version 3. `https://www.sqlite.org/datatype3.html`, 2017.

[36] The SQLite Authors. File Locking And Concurrency In SQLite Version 3. `https://www.sqlite.org/lockingv3.html`, 2017.

[37] L. Toka, M. Dell'Amico, and P. Michiardi. On scheduling and redundancy for P2P backup. *CoRR*, abs/1009.1344, 2010.

[38] Tokio Project. Tokio - a platform for writing fast networking code with rust. `https://tokio.rs/`, 2017.

[39] Tokio Project. Tokio - streaming protocols. `https://tokio.rs/docs/going-deeper-tokio/streaming/`, 2017.

[40] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[41] Wikipedia. Birthday attack - Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Birthday_attack&oldid=797537980`, 2017.

[42] Wikipedia. ext4 - Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Ext4&oldid=809382067`, 2017.

[43] A. Young and M. Yung. Cryptovirology: extortion-based security threats and countermeasures. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 129–140, May 1996.

List of Figures

List of Tables

Glossary

administrator

A person that is in charge of managing the system (e.g. add / remove nodes).

chunk

A piece of data in the system consisting of binary data (*chunk content*), a unique identifier (*chunk identifier*) and an *expiration date*.

chunk content

The binary data a chunk represents.

chunk identifier

A unique identifier of a *chunk* that is derived from the corresponding *chunk content*, e.g. using hash functions.

chunk index

A *client*-side data structure that stores file attributes, folders and file to *chunks* mappings.

chunk table

A data structure on a *node* containing all *chunk identifier* and *expiration dates* of all *chunks* managed by this *node*.

client

A piece of software that runs on a users computer, in charge of creating and restoring backups (see A.4.3.1 Client).

client m-replication

The *client* defines a custom degree of redundancy from 1 to the number of *nodes* in the system. See 2.2 Fundamental Design Decisions.

designated node

A *node* that is selected by a *client* to send backup data to or restore data from.

expiration date

Date, until which a given *chunk* must be kept in the system.

file

A document on a client system that shall be backed up. A file is represented in the redbackup system by one or more *chunks*.

header field

One piece of meta information that is part of a *message header*, consisting of a key and value.

leaving node

A *node* that is scheduled to leave the system permanently.

location

Nodes can be associated with a (physical) location to ensure redundancy is met over multiple physical locations.

management

A component that orchestrates the configuration of the system (see A.4.3.4 Management).

medium

Particular form of storage for files, e.g. hard disks or magnetic tape.

message

A unit of communication between two parties (see A.4.5 Messages).

message header

Meta information of a *message* that define operating parameters, consisting of multiple *header fields*.

message payload

Actual data of a *message* (see A.4.5 Messages).

new node

A *node* scheduled to be integrated into the system, that was not part of that or any other system before.

node

A core participant in the system that manages *chunks* (See A.4.3.2 Node).

node identifier

A unique identifier assigned to any node to distinguish it from other *nodes*.

node metadata

A set of data that describes a *node* (e.g. its state, storage capacity). The set of data differs depending on the context.

node state

The intrinsic or extrinsic state of any node (see A.4.3.2 Node).

node-cache

A list of contact information of nodes in the system that each client buffers in case the management is unavailable.

root handle

A flag to mark (the first part of) a *serialised chunk index* to find all chunks associated with a backup for a restore.

sending node

A *node* that is sending data to a client or another node.

serialised chunk index

A serialised version of the *chunk index*, that might be split into multiple chunks, created on backup and used for restore.

storage

Component in charge of persisting data (see A.4.3.3 Storage).

system

The whole redbackup system as described in A.4 Architectural Concept Paper.

system m-replication

Nodes replicate *chunks* to m other *nodes*. The degree of redundancy is defined globally by the administrator. See 2.2 Fundamental Design Decisions.

system n-replication

Nodes replicate *chunks* to all other *nodes*. As a result, the degree of redundancy is equal to the number of *nodes* in the system. See 2.2 Fundamental Design Decisions.

user

A registered user with valid credentials who wants to backup and restore its data. Has a client installed on his local computer.

Appendices

A.1 Task Description

redbackup: a redundant distributed backup system prototype

Fabian Hauser
Raphael Zimmermann

December 11, 2017

1 Client and Supervisor

Supervisor: Prof. Dr. Farhad Mehta, HSR Rapperswil

Client: IFS (Institut fr Software), HSR

2 Students

- Raphael Zimmermann, rzimmerm@hsr.ch
- Fabian Hauser, fhauser@hsr.ch

3 Setting

Today, most individuals, as well as small and medium enterprises (SME), create backups with local devices (mostly external hard disks), which are not physically distributed[1]. Alternatively, some SME, and individuals store their data online using cloud storage providers, but legal issues, privacy concerns and the dependency on large providers limit the widespread use of cloud storage. Currently, the security of cloud backups is based solely on the strength of cryptographic algorithms and security measures taken by cloud providers.

Automated backup solutions, which store the data in a decentralised and distributed manner within a trusted network could solve this issue. Today, however, there is no ready to use solution available on the market.

4 Vision

A possible solution for a efficient, decentralised and distributed backup could be realised with a combination of two parts,

1. a distributed storage system, which provides redundant, deduplicated storage and
2. an easy to use client software, which creates encrypted backups and uploads the data into the distributed system.

To permit a widespread usage of such a solution, it is crucial that both parts are easy to install and configure.

[1]The issue of local backups was brought to broad public attention in may 2017 due to wide spread infections of the ransomware "WannaCry" that also encrypted files on attached devices and network shares. See [3, 2]

The distributed system as proposed above (1) should be able to:

a. build a peer-to-peer network of trusted nodes over the internet.

b. handle joining of nodes including authentication.

c. handle planned and unplanned leaving of nodes.

d. store data with a defined degree of redundancy.

e. protect data from being modified or deleted[2].
The system may discard data after it has reached a specified expiration date.

f. address and retrieve stored data.

g. allow access from clients including authentication over an API.

The client software as proposed above (2) should be able to:

1. create backups of files and directories (content and metadata).
2. compress and encrypt backups.
3. upload backups into the distributed system (see 1).
4. retrieve, decrypt, decompress and recover data from the distributed system (see 1).

5 Goals

The goal of the Study Project is to provide a theoretical description of an append only, distributed peer-to-peer data storage as the basis for the described vision, as well as a working prototype.

The theoretical concept must address the following issues:

a. joining of nodes

b. planned and unplanned leaving of nodes

c. distribution of data with a defined degree of redundancy

d. fair distribution of data blocks within the distributed system

e. uploading data into the distributed system

f. addressing within the distributed system

g. retrieving stored data

h. scalability for up to several 100 nodes where every node can store a data volume of up to 2 terabytes.

i. detection of inconsistent and unresolvable states (e.g. redundancy failures because there are not enough nodes)

All the above problems will be implemented in a prototype to demonstrate the described concepts. It is not intended for production usage. The concept, as well as the prototype, assumes that all nodes are addressable by a fixed IP-address and port. Encryption and deduplication of data on the client will not be discussed within this study project since it is limited to uploading and retrieval of data.

A short evaluation will be carried out at the beginning of the project in order to find an appropriate implementation language for the prototype. The current candidates for a suitable implementation language are Rust and Erlang.

[2]The idea of an append only approach prevents accidental or malicious deletion of stored data, for example by randsomware.

6 License

The study project is inteded to be free software and will be published under the AGPL-License [1].

7 Guidelines

The students and the supervisor will plan weekly meetings to check and discuss progress.

All meetings are to be prepared by the students with an agenda. The agenda will be sent at least 24h prior to the meeting. The results will be documented in meeting minutes that will be sent to the supervisor.

A project plan must be developed at the beginning of the thesis to promote continuous and visible work progress. For every milestone defined in the project plan, the temporary versions of all artefacts need to be submitted. The students will receive a provisional feedback for the submitted milestone results. The definitive grading is however only based on the final results of the formally submitted report.

8 Documentation

The project must be documented according to the regulations of the Computer Science Department at HSR (see `https://www.hsr.ch/Allgemeine-Infos-Bachelor-und.4418.0.html`). All required documents are to be listed in the project plan. All documents must be continuously updated, and should document the project results in a consistent form upon final submission. The documentation has to be completely submitted in 3 copies on CD/DVD. A printed version has to be delivered if requested by the supervisor.

9 Important Dates

Refer to `https://www.hsr.ch/Termine-Diplom-Bachelor-und.5142.0.html`.

10 Workload

A successful study project results in 8 ECTS credit points per student. One ECTS points corresponds to a work effort of about 30 hours. All time spent on the project must be recorded and documented.

11 Grading

The HSR supervisor is responsible for grading the study project. The following table gives an overview of the weights used for grading.

Facet	Weight
1. Organisation, Execution	1/5
2. Report	1/5
3. Content	3/5

The effective regulations of the HSR and Department of Computer Science apply (see `https://www.hsr.ch/Ablaeufe-und-Regelungen-Studie.7479.0.html`).

Rapperswil, December 2017

Farhad Mehta

Prof. Dr. Farhad Mehta
12.12.2017

References

[1] Gnu affero general public license. `https://www.gnu.org/licenses/agpl-3.0.en.html`.

[2] M. Scott and N. Wingfield. Hacking Attack has security experts scrambling to contain fallout. `https://www.nytimes.com/2017/05/13/world/asia/cyberattacks-online-security-.html`, 2017.

[3] J. C. Wong and O. Solon. Massive ransomware cyber-attack hits nearly 100 countries around the world. `https://www.theguardian.com/technology/2017/may/12/global-cyber-attack-ransomware-nsa-uk-nhs`, 2017.

A.2 Project Plan

Redbackup: Project Plan

Authors:
Fabian HAUSER and
Raphael ZIMMERMANN

Advisor:
Prof. Dr. Farhad MEHTA

Autumn Term 2017

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Contents

Chapter 1

Project Overview

The goal of the study project is to provide a theoretical description of an append-only, distributed peer-to-peer data storage as well as a working prototype as described in the problem statement [3].

Chapter 2

Project Organization

All team members have the same strategic rights and duties. Prof. Dr. Farhad Mehta is our project advisor as visible in Figure 6.1.

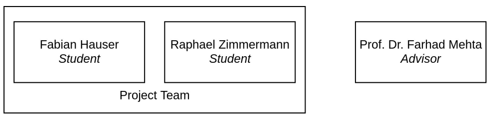

FIGURE 2.1: Project organization chart

2.1 Roles

Due to the small team size, most roles are performed by both team members.

Raphael Zimmermann project management, software engineering, quality assurance.

Fabian Hauser infrastructure management, software engineering, quality assurance.

Chapter 3

Project Management

3.1 Components

For a better overview and to allow us a sophisticated time assessment, we decided to group tasks into categories, i.e. JIRA components. Components represent processes, documents and products which are to be released.

Currently, tasks are separated into following components:

- Final Submission Document
- Concept Paper
- Management
- Poster
- Presentation
- Project Plan
- Prototype

3.2 Time Budget

The project started with the Kickoff Meeting on 18.09.2017 and will be completed after 14 weeks by 22.12.2017. The two team members are available for 240 hours each during this period which corresponds to a weekly time budget of 17.15 hour per person.

Apart from the statutory holidays, there are no further absences planned. Statutory holidays do not affect the weekly time budget.

3.3 Schedule

The project schedule is an iterative process based on elements of SCRUM.

Due to the explorative nature of the project, we decided on a sprint duration of one week.

3.3.1 Iterations & Milestones

The goals and milestones resulting from each sprint are shown in Figure 3.2.

3.3.2 Deviations from Original Schedule

The tasks and milestones were continuously updated during the project course, as is evident from Figure 3.2. The most notable additions and exclusions are noted hereafter.

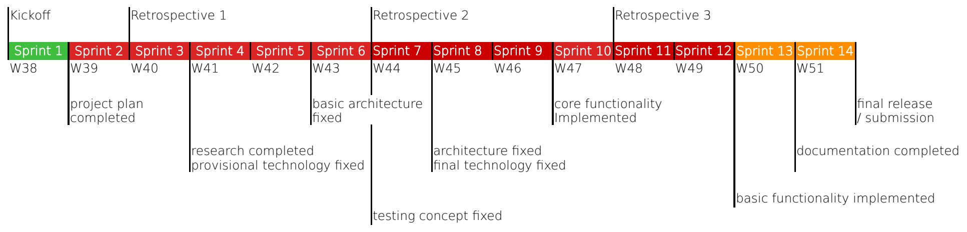

FIGURE 3.1: Overview of the 14 iterations

Research Report After the underlying research at the project start, we decided to not further research and summarise possible data distribution mechanisms, as it would not have been possible to finish the report and project in time otherwise.

Rust Programming Language As we settled for the Rust programming language in the language evaluation, we introduced the additional task to get familiar with rust an its network stack.

Retrospective 2 During the last standup meeting in sprint 6, we decided to not hold the retrospective meeting 2, as we both had no open topics to discuss.

Presentation We decided to not create a presentation during the project course as this is no mandatory part of the study project.

3.3.3 Meetings

The team works every Tuesday (10:00 - 17:00) together in the study project room and every Monday (08:00 - 17:00) remotely. Both days begin with a daily stand-up meeting taking no longer than 15 minutes. Sprint planning meetings are carried out on Tuesday at 10:00. Table 3.1 shows an overview of the total meeting time budget.

TABLE 3.1: Meeting Time Budget

Meeting Type	Total Duration per Person	Total Duration for the Team
Supervision Meetings	10 hours	20 hours
Standup Meetings	7 hours	14 hours
Sprint Planning Meetings	16 hours	32 hours
Retrospective	3 hours	6 hours
Total	36 hours	72 hours

Regular meetings with the project advisor take usually place on Friday in Prof. Dr. Mehta's office. The agenda must be sent via email to all participants at least 24 hours prior the meeting.

Raphael Zimmermann will take meeting notes for every meeting. Meeting minutes are published on the project website afterwards.

Sprint	Sprint 1	Sprint 2	Sprint 3	Sprint 4
Tag	0.1.0	0.2.0	0.3.0	0.4.0
Date	19.09.2017-25.09.2017	26.09.2017-02.10.2017	03.10.2017-09.10.2017	10.10.2017-16.10.2017
Milestones	~~- project plan completed~~	*- project plan completed*	- provisional technology fixed - research completed	
Time Budget	34	34	34	34
Tasks	- write project plan - adapt Infrastructure	- academic research - brainstorm basic concept - define quality of service attributes - retrospective 1 (at the end of the sprint)	- write basic concept (including testing concept) - carry out language evaluation ~~- academic research~~	- draft basic architecture & domain model - bootstrap project ~~- *prototype architectural critical components*~~ *- get familiar with Rust and its Network Stack*
Documents / Artefacts	- problem statement - project plan draft	- first draft of concept paper - project plan finalized	~~- research report~~ - language evaluation - second draft of concept paper	- third draft of concept paper with drafted architecture

Sprint	Sprint 5	Sprint 6	Sprint 7	Sprint 8
Tag	0.5.0	0.6.0	0.7.0	0.8.0
Date	17.10.2017-23.10.2017	24.10.2017-30.10.2017	31.10.2017-06.11.2017	07.11.2017-13.11.2017
Milestones	- basic architecture fixed	- testing concept fixed	- architecture fixed - final technology fixed	
Time Budget	34	34	34	34
Tasks	~~- *prototype architectural critical components*~~ *- get familiar with Rust and its Network Stack*	*- prototype architectural critical components* - prototype testing architecture ~~- *finalize testing concept*~~ ~~- retrospective 2~~ *(Skipped)*	- finalize architecture *- finalize testing concept* *- implement core functionality*	- implement core functionality - set up testing infrastructure
Documents / Artefacts	- fourth draft of concept paper with provisional architecture	- fifth draft of concept paper with final testing concept	~~- *concept paper*~~	*- concept paper*

Sprint	Sprint 9	Sprint 10	Sprint 11	Sprint 12
Tag	0.9.0	0.10.0	0.11.0	0.12.0
Date	14.11.2017-20.11.2017	21.11.2017-27.11.2017	28.11.2017-04.12.2017	05.12.2017-11.12.2017
Milestones	- core functionality implemented			- basic functionality implemented
Time Budget	34	34	34	34
Time Spent	33.25	29.25	28	
Tasks	- implement core functionality	- implement basic functionality - retrospective 3 (at the end of the sprint)	- implement basic functionality - exhaustive testing	- improve code quality & documentation - exhaustive testing - performance testing *- begin with final submission document*
Documents / Artefacts				

Sprint	Sprint 13	Sprint 14
Tag	0.13.0	0.14.0
Date	12.12.2017-18.12.2017	19.12.2017-22.12.2017
Milestones	- documentation completed	- final release / submission
Time Budget	34	34
Tasks	- finalize release documents - prepare presentation	- prepare submission archive (e.g. JIRA export) - print documents
Documents / Artefacts	- management summary - abstract - final submission document ~~- *poster*~~ ~~- *presentation*~~	- final submission archive (CD) *- poster*

FIGURE 3.2: Detailed overview with tasks and milestones of all 14 sprints. *Italic* and ~~struck out~~ text mark deviations from the original project plan that occurred during the projects course.

Chapter 4

Risk Management

An assessment of the project-specific risks is carried out in Table 4.2 as time loss during the whole project. The risk matrix in Table 4.1 provides an overview of the risk weighting.

To account for these risks, we reduce our weekly sprint time by the total weighted risk applicable to the planned task topics (on average approximately 20%). We also review the risk assessment after every sprint, adapt it and take measures if necessary.

Severity / Probability	High ($\geq 5d$)	Medium (2-5d)	Low ($\leq 2d$)
High ($\geq$ 60%)	1	2	
Medium (30-60%)	8	3, 4	
Low ($\leq$ 30%)		5, 6, 7	

TABLE 4.1: The risk matrix. Numbers reference to the risk assessment Table 4.2

TABLE 4.2: Risk assessment table. Time in hours over the total project duration.

#	Title	Description	Prevention / Reaction	Risk [h]	Probability	= [h]
1	Problems with technology stack	Parts of the selected technology stack are not well suited, incomplete or immature.	Reflect the suitability of the chosen technology during the architecture draft.	60	60%	36
2	Missing aspects in concept paper	The concept paper does not fully cover all necessary aspects of the prototype.	Invest an adequate amount of time in architecture design.	30	60%	18
3	The architecture/concept does not scale.	The chosen architecture/concept does not scale to the expected data volume or node size	Invest an adequate amount of time in architecture design.	30	40%	12
4	Communication errors	Errors due to miscommunication or misapprehension.	Maintain a high level of interaction, precise specification of tasks responsibilities, conduct meetings if ambiguities exist.	20	50%	10
5	Problems with project infrastructure	The used project infrastructure is not or only partially available, or data loss occurs within management software.	Clean setup and self-hosting of the tools to prevent third-party dependencies.	30	30%	9
6	Scope creep	The project's scope is extended over the project course.	Define the project scope and limitations precisely. Discuss changes with the project advisor.	30	30%	9
7	Dependency errors	There are errors/bugs in third-party dependencies, i.e. libraries.	Carefully select libraries and limit thid-party dependency to a minimum.	20	30%	6
8	Missing dependency documentation	Selected libraries are lacking proper documentation	The documentation quality of a library should be a selection criterion.	15	40%	6
	Total weighted risk					106

Chapter 5

Infrastructure

5.1 Project Management and Development

For project management, document/code storage and continuous integration/deployment we utilise the corresponding products by Atlassian (JIRA, BitBucket, Bamboo, Crowd)[1].

These applications are hosted on our HSR project server (*sinv-56017.edu.hsr.ch*), which runs a standard Ubuntu Linux 17.04.

5.1.1 Development Tools

Since most development tools depend on the chosen technology and might change in the future, they are maintained separately on the project website [1].

5.2 Backup and Data Safety

An incremental backup of the project server including the source code and documentation is created on an independent system (*pin1262031.hsr.ch*) every night.

As our documents and code is stored in a git repository, they are also distributed on all development systems.

[1]https://www.redbackup.org/development/

Chapter 6

Quality Measures

To maintain a high standard of quality, we take the following measures:

- short sprint reviews
- three extended retrospectives
- code reviews
- automated unit and integration testing
- publish all documentation on the project website using continuous integration/delivery.
- using continuous integration for source code

6.1 Documentation

The official documents such as the final submission document, the theoretical concept as well as this project plan are written in LaTeXand published on the project website [1] as PDF documents. All other documents such as meeting minutes are written in Markdown and made available in HTML on the website as well.

The sources are in both cases kept under version control in the same repository, which allows us to use the same tools and processes for documentation and code. The continuous integration server builds and publishes the website whenever new changes are pushed to the repository.

6.2 Project Management

Because the project plan allows for an iterative process, we use JIRA with its SCRUM-Features (such as sprint creation or boards) for project management.

6.2.1 Sprint Planning

Each sprint is mapped to JIRA, which allows the project advisor to trace the project progress. Sprints are represented as boards on which the current state and assignee of any issue is easily visible ("To Do", "In Progress", "Review", "Done").

6.2.2 Definition of Done / Review of Pull Requests

An issue may be closed if ***all*** of the following conditions are met:

- All functionality conforms to the specification. Any deviations must be discussed and decided by the team.

[1] https://www.redbackup.org

- A review is performed and accepted in a pull request.
  - The source code is reasonably documented.
  - No code is commented out.
  - No warnings and errors by the compiler or any other quality tool.
  - Reasonable unit and integration tests exist and pass.
  - All documentations are up to date including the project website.
  - The complete continuous integration pipeline works.
  - The code is formatted according to the guidlines (i.e. according to RustFmt)
- The corresponding branch is merged into the stable branch (e.g. master).
- All time is logged.

6.3 Development

Since we are in a very early and agile stage we decided to use GitHub Flow[2], a straightforward development workflow.

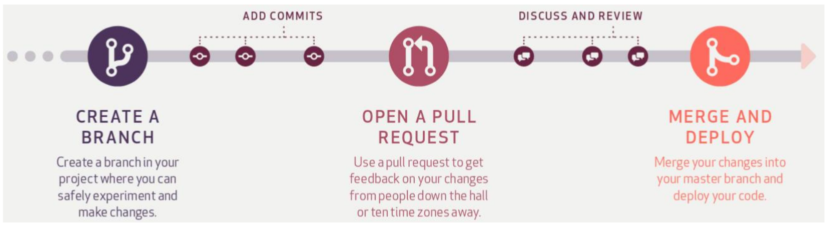
ADD COMMITS
DISCUSS AND REVIEW
CREATE A BRANCH
Create a branch in your project where you can safely experiment and make changes.
OPEN A PULL REQUEST
Use a pull request to get feedback on your changes from people down the hall or ten time zones away.
MERGE AND DEPLOY
Merge your changes into your master branch and deploy your code.

FIGURE 6.1: GitHub Flow illustrated (Source [2])

Since the effective technology will be fixed later in the project, concrete coding guidelines, tools, metrics and an error policy will be defined when appropriate.

6.4 Testing

All functionality must be automatically testable using continuous integration. Any non-trivial function/method must be verified with unit tests.

Integration tests verify extended test scenarios.

A minimal performance analysis will be carried out at the end of the project.

Bibliography

[1] Atlassian Inc. Open Source Services by Atlassian Inc. https://developer.atlassian.com/opensource/, 2017.

[2] Github Inc. Github Flow. https://guides.github.com/introduction/flow/, 2013.

[3] F. Hauser and R. Zimmermann. Task Description "redbackup: a redundant distributed backup system prototype". https://www.redbackup.org/problem-statement.pdf, 2017.

List of Figures

List of Tables

A.3 Requirements

The System knows two roles: ***Users*** and ***Administrators***.

A typical ***User*** does not want to interact with the system at any time. All he/she wants is to be sure that all his/her data is safely backed up.

An ***Administrator*** wants to interact with the system as seldom as possible. He/She wants to be sure that the system runs smoothly. If something goes wrong, he/she wants to be able to fix it within a few minutes.

Because the actors can have very different levels that they tolerate it is hard to determine measurable requirements. We, therefore, started to highlight the intentions and general attitude of the actors towards the system. Afterwards, we extrapolated the most important requirements.

Not all intentions must be met within the study project but the might impact architectural decisions. All intentions relevant for the study project are therefore listed here ***emphasised***.

A.3.1 Intentions of a User

As a user, . . .

on the topic of **backup creation**, . . .

1. I want my backups to take place automatically in the background so I do not forget.
2. I want to get notified if the creation of a backup fails, but not for every little trivia. For example, if the network connection breaks down temporary, the process should be retried several times before showing a notification.
3. I want to get notified if I have not created a new backup for a suspicious amount of time (e.g. usually daily but suddenly no backups for a week).
4. I want that the creation of a backup does not impact the performance of my device noticeably (CPU, RAM, network etc).
5. I want to be able to stop/restart/suspend my device at any time even if a new backup is in the process of being created.
6. I want to exclude certain files from being backed up, for example, downloads.

on the topic of **backup storage**, . . .

7. I want my backup data to be encrypted so that only I can restore it.
8. I want to create backups any place with a working internet connection (especially not just from home).
9. ***I want my backup data to be stored only on physical sites that I trust.***
10. ***I want my backups to be space efficient to save cost.***
11. ***I want my data to be stored with a defined degree of redundancy so that it does not get lost.***
12. ***I want my data to be replicated to different places (e.g. buildings) in case one of them is subject to a catastrophic event (e.g. an earthquake, fire).***

13. ***I want that my data is protected from unauthorised access at any time***
14. ***I want that my data is not lost if a storage medium is corrupted or breaks down.***
15. ***I does not matter, where my data is replicated to within my trusted network.***
16. ***I don't care where within the trusted network my data is backed up to as long as the creation and restore of backups does not take 30% longer as if it were stored within the local network.***
17. ***I want that the duration of a backup is linearly dependent of the size of the data that has changed.***
18. ***I want to define how long backups are stored or use a reasonable default.***

on the topic of **backup restore**, . . .

19. ***I want to be able to restore my data as easily as possible in case of a catastrophic event.***
20. ***I want to be able to restore a previous version of a (possibly deleted) file within less than five minutes.***
21. I want to see the progress when I restore data so that I can estimate when it is done.

on the topic of the **backup software**, . . .

22. I want be able to install and have the software up and running within five minutes.
23. I don't want to perform updates manually.

A.3.2 Intentions of an Administrator

As an administrator, . . .

on the topic of **backup system management**, . . .

1. ***I want to grant new users access to the backup system.***
2. ***I want be able to see how much capacity a user is using.***
3. ***I want to limit the capacity per user.***
4. I want to define default profiles on how long to keep backups for new users.
5. ***I want to extend the storage capacity by plugging in a new disk and starting/-connecting a new node.***
6. ***I want to add new backup sites (and therefore nodes) to the system.***
7. ***I want to scale the system up to 100 Nodes and down to 2.***
8. ***I want to put multiple large disks (e.g. 2TB) into a node so that I do not have to maintain an unjustifiably high number of nodes.***

9. ***I want the system to operate properly if the management node is down (no single point of failure). The addition of new nodes may not be possible without management.***

10. ***I want to perform updates including reboots without impacting the performance of the network.***

11. ***I want to update the management server (including restart) without impacting the system (except for the configuration/monitoring part).***

on the topic of **backup monitoring**, . . .

12. ***I want to receive notifications quickly (i.e. within an hour) when a disk or node in the System fails.***

13. ***I want to receive notifications within an hour if a disk or node might fail soon (e.g. SMART-results indicate that a disk will fail).***

14. ***I want to be informed if a certain replication-degree cannot be met.***

A.3.3 Requirements

The following requirements are grouped according to ISO 9126 [14].

A.3.3.1 Functionality

1. ***A node must guarantee when asked about files stored on it that all concerned files exist on it and are not corrupted*** (accuracy)

2. ***A given node must only know the minimal amount of data required to detect corrupted data and garbage collection, i.e. the (encrypted) binary data, an identifier that can be derived from the encrypted binary data as well as the expiration date.*** (security)

3. An administrator must be able to grant new users access to the backup system within less than 10 minutes. (suitability)

4. An administrator must be able to see how much capacity in GB a given user is using within less than 1 minute. (suitability)

5. All data sent from a user must be encrypted according to BSI TR-02102-1 [3] (security)

6. Nodes and users that are not explicitly listed by an administrator are not allowed to participate in a given system and can therefore under no circumstances read or write data from or in the system. (security)

A.3.3.2 Reliability

7. ***Data must be replicated on at least two nodes in the system 1 hour after its arrival in the system.*** (fault tolerance)

8. ***A storage must periodically verify the integrity of the data stored in it so that corruptions are detected after max. 24 hours.*** (fault tolerance)

9. ***The replication process is not affected if the management cannot be reached. The only allowed exceptions are the addition of new nodes and planned leaving.*** (recoverability)

10. ***If a user/administrator updates and reboots a client/node/management so that it is not down for more than 10 minutes, no notifications/warnings are sent nor must it trigger any recovery routines.***(recoverability)

11. An administrator must receive a notification via email within 10 minutes in case a disk, node or management fails. (maturity, fault tolerance)

12. An administrator must receive a notification via email within 10 minutes if a potential disk failure (using SMART results) is detected. (maturity, fault tolerance)

13. An administrator must receive a notification via email within 10 minutes if the replication-degree for a file/chunk cannot be met. (maturity, fault tolerance)

A.3.3.3 Usability

14. ***A user must be able to inspect the current progress of a backup/restore process in percent, time spent as well as an estimate for the remaining time*** (understandability).

15. ***A user or administrator must be able to install the client/node/management software including all its dependencies within less than 5 minutes (excluding the time spent for downloads).*** (operability, attractiveness)

16. ***A user must be able to define ignored files within less than 3 minutes.*** (operability, attractiveness)

17. ***A user must be able to restore a previous version of a (possibly deleted) file within less than five minutes if it is file size is less than 100MB.*** (operability, attractiveness)

18. If a user pauses (e.g. system restart, suspension) his/her device during backup and restore, the client software must continue its work where it stopped. (operability, attractiveness)

A.3.3.4 Efficiency

19. ***The time for the creation of a backup must be*** $O(n)$ ***where*** n ***is the amount of changed data bytes.*** (time behaviour)

20. The creation of a backup on the client should use more than 10% of the systems resources (i.e. CPU and RAM). (resource utilization)

21. A node must be able to persist data on more than two 2TB disks. (Resource utilization)

A.3.3.5 Maintainability

22. ***Every interaction of a client/node/management including all their side effects (e.g. the persistence of data, sending data over the network) must be logged so that it in case of a failure every step can be reproduced.*** (analysability)

23. ***The hash-Algorithm used for the calculation of identifiers must be replaceable in the code within one working day.*** (changeability)

24. ***A user/administrator must be able to change the configuration on a client/node/management using a configuration file.*** (changeability)

25. ***All components must be testable with unit tests as well as integration tests to cover at least 80% of all lines.*** (testability)

26. Any component (i.e. client, node, management) must not depend on any other component except sending messages so that it is possible to replace it with a version written in another technology. (changeability)

27. Nodes/Clients must be able to upgrade chunk-Identifiers to a new hash-Algorithm in $O(n)$ where n is the amount of chunks. (changeability)

A.3.3.6 Portability

28. ***All components must be published as docker images to simplify the deployment in docker environments***. (installability)

29. A user must be able to export and decrypt all his/hers data in $O(n)$ where n is the size of all his/hers binary data in order to switch to another backup solution (replaceability)

A.4 Architectural Concept Paper

A.4.1 Overview

Figure 1 presents an overview over the Redbackup *system*. All actors, components and interactions are described in detail in the following sections.

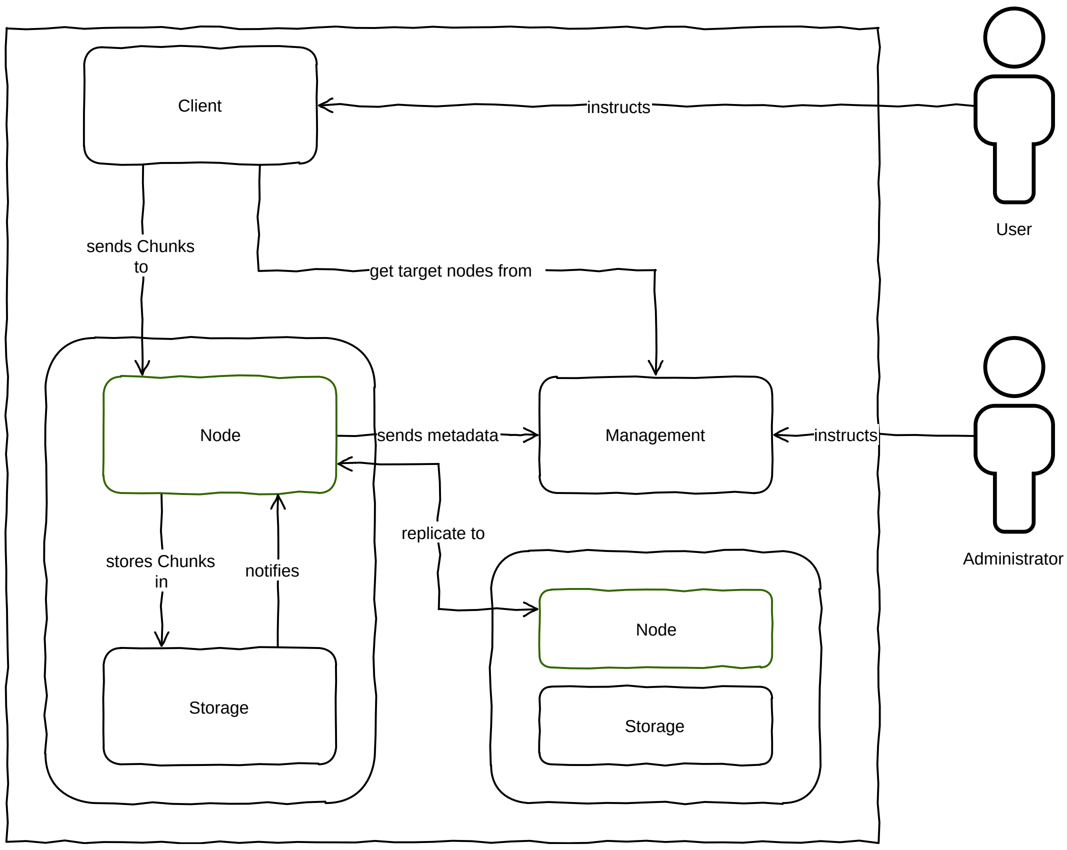

FIGURE 1: A simplified *system* overview

A.4.2 Actors

A.4.2.1 User

A typical *user* does not want to interact with the *system* at any time. All he/she wants is to be sure that all his/her data is backed up.

To simplify the implementation for the study project, the *user* must instruct the *client* manually (create and restore backup).

Responsibilities

- Instruct the *client* to create a Backup
- Instruct the *client* to restore a Backup

A.4.2.2 Administrator

An *administrator* wants to interact with the *system* as seldom as possible. He/She wants to be sure that the *system* runs smoothly. If something goes wrong, he/she wants to be able to fix it within a few minutes.

Responsibilities

- Instruct the *management* to add/remove *nodes* from the *system*
- Ensure that the *management* runs properly (e.g. monitor it)
- Act if the *management* notifies him/her about any anomalies

A.4.3 Components

A.4.3.1 Client

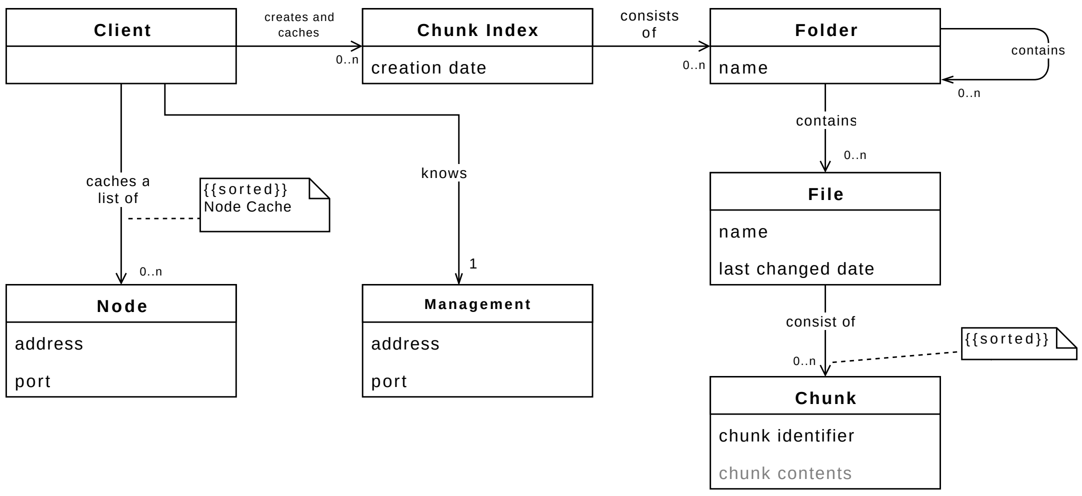

FIGURE 2: Subdomain illustrating the *clients* perspective

- The *client* knows the *management* by configuration.
- The *client* queries the *management* for a list of *nodes*, to which he can send backups or restore previous backups. He caches the results in its *node-cache*.
- The *chunk index* hierarchically models the file system. This structure might change in the future when supporting symlinks and permissions [23].
- The relationship between a given *file* and its *chunks* is essential. The *client* splits a *file* into multiple *chunks* to speed up the backup of large data. During the study project, a *file* consists of exactly one *chunk*.
- A *chunk* is identified by its *chunk identifier*, which can be calculated using a hash function which takes the *chunk contents* as input.
- The *chunk contents* do not have be present on the *client*. During backup, the *client* can calculate the *chunk contents* and the corresponding *chunk identifier* from the *files* on the disk. On restore, the *client* fetches the *chunk contents* from a *node* and reassembles the *file* based on the *chunk index*.

Responsibilities

- Keeping the *node-cache* up to date
- Create Backups (See Scenario A.4.4.1: Create Backup)

  - Building/Calculating the *chunk index*
  - Serialize the *chunk index* into *chunks* including a *root handle*
  - Send *chunks* to *nodes*
  - Mark *chunks* as *root handles*

- Restore Backups (See Scenario A.4.4.2: Backup Restore)
  - Fetch *root handles* from *nodes*
  - Fetch *chunks* from *nodes*
  - Deserialise *chunk indexs*
  - Reassemble *chunks* into *files* (not required for the study project)
  - Let the *user* choose which *files* from which *chunk index* shall be restored.

A.4.3.2 Node

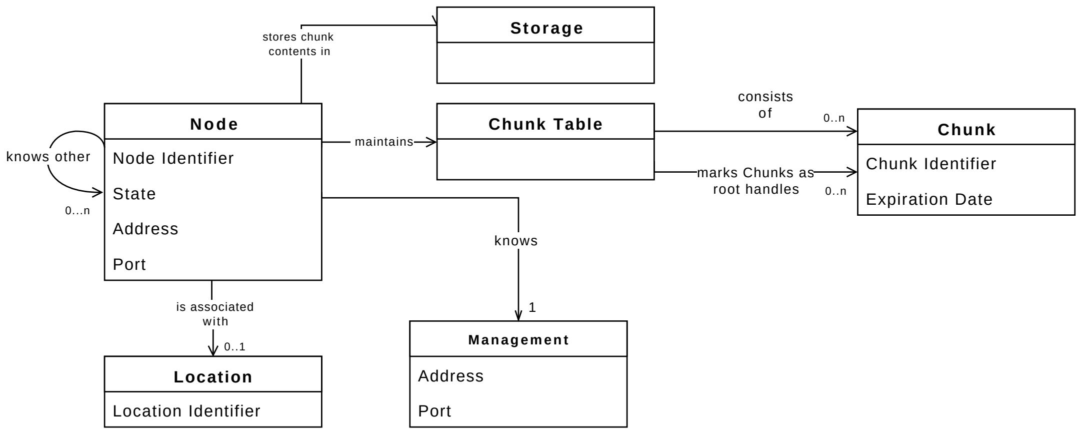

FIGURE 3: Subdomain illustrating the *nodes* perspective

- A *node* is identified by its unique *node identifier*. It can be addressed using a certain ***Address*** and ***Port***.
- A *node* is also associated with a *location* which will be a replication criteria in the future.
- A *node* knows the *management* by configuration and exchanges data with it on a regular basis (Messages ***get nodes metadata*** and ***post nodes metadata***)
- A *node* is in a given *node state* and acts differently depending on it (See Figure 4).
  - ***uninitialized***: The *node* queries the *management* for initialization and does nothing else.
  - ***participating***: This is the "normal" *node state* in which a *node* accepts backups, performs replication and sends requested data.
  - ***unreachable***: A *node* is marked as ***unreachable*** if any other *node* or the *management* can not reach it. A *node* never sees itself in this *node state*.

  - ***leaving soon***: The *node* does not answer any requests and ensures that all its data is replicated. When it is done, it automatically switches into the ***left*** *node state*.
  - ***left***: The *node* has nothing to do in this *node state* and can shut down.

- A *node* knows all other *nodes* in the *system* and maintains a *node state* (see Figure 4) for them as well.
- The *chunk contents* are stored in a *storage*. See A.4.3.3 Storage for further details.
- Which *chunks* are stored on the *node*, their *expiration date* and the information whether a *chunk* is a *root handle* is stored in the *chunk table*.

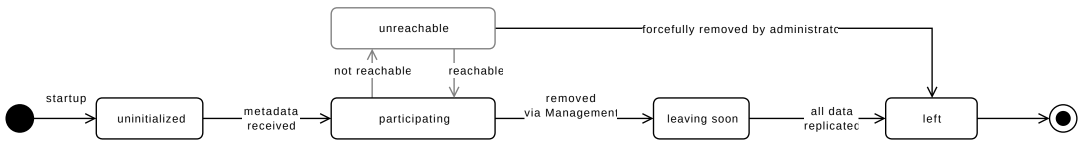

FIGURE 4: A UML State Diagram describing the *node states*

Responsibilities

- Send *node metadata* periodically (Messages ***get nodes metadata*** and ***post nodes metadata***) in order to ...
  - Perform initialization
  - Learn about other *nodes*
  - Start the ***leaving*** process and notify the *management* when it is completed
  - Send *node metadata* to the *management* so that it can perform health-checks (e.g. verify the timestamp)
- Maintain the *chunk table*
  - Update *expiration dates*
  - Add new *chunks*
  - Remove expired *chunks*
- Handle possible *storage* issues (see Scenario A.4.4.8 Data Storage has Errors)
- Reply to ***get designation*** (Scenario A.4.4.1 Create Backup) , ***get root handles*** (Scenario A.4.4.2 Backup Restore) and ***get chunks*** requests
- Replicate *chunks* (See Scenario A.4.4.6 Data Replication)
- Handle ***leaving*** process (see Scenario A.4.4.4 Node Leaving Planned)

A.4.3.3 Storage

- The main purpose of the *storage* component is to persist *chunk contents*. A *storage* is associated with one *node* which stores, loads and deletes *chunk contents* by its *chunk identifier* on the *storage*. The same *node* is notified by the *storage* e.g. when corrupted data is detected.

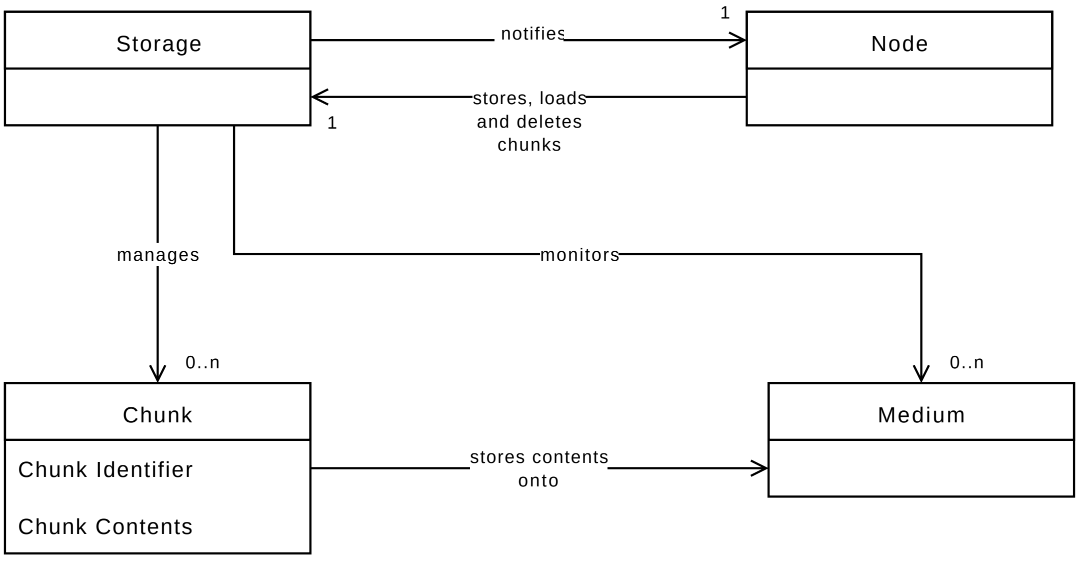

FIGURE 5: Subdomain illustrating the *storage*s perspective

- A *chunk* should also monitor the *mediums*, on which the *chunk contents* are stored, for possible issues and report them to the *node*. This feature, however, is not implemented in the study project.
- The *storage* component is deployed on the same host as the *node* that is using it.

Responsibilities

- Store *chunk contents*
- Load and return the *chunk contents* for a given *chunk identifier*
- Delete the *chunk contents* for a given *chunk identifier*
- Detect possible corruption of *chunk contents*
- Perform service checks on the *mediums* (e.g. S.M.A.R.T tests)
- Notify the *node* when a corruption / integrity problem occurs

A.4.3.4 Management

- The *administrator* is the only Actor interacting with the *management*. He/She sends instructions (e.g. add new *node*) and is notified by the *management* about any anomalies.
- *clients* fetch a list of all *nodes* from the *management* (See Component A.4.3.1 Client for further details). As for the study project, the *management* does not have any information about *clients*, but this might change when authentication is implemented in the future.
- The *management* maintains *node metadata* for each *node* in the *system* and sends that information every *node* when they post their *node metadata*.

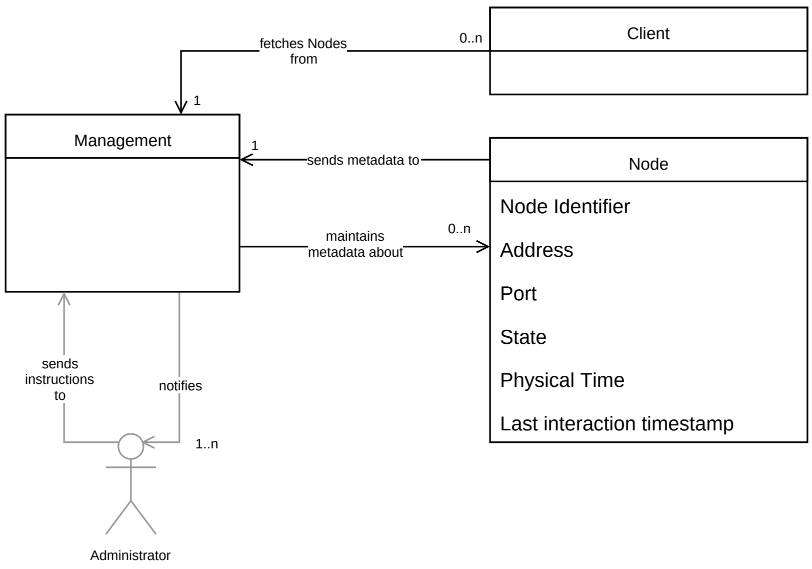

FIGURE 6: Subdomain illustrating the *management*'s perspective

Responsibilities

- Add and remove *nodes*
- Notify the *administrator* e.g. if a *node* leaves unexpectedly.
- Receive *node metadata* from the *nodes* and reply with *node metadata*
- Send list of *nodes* to the *clients*

A.4.4 Scenarios

This section describes various scenarios of *system* usage and possible failures.

In the sequence diagrams, we use the composition of the minimal integration test, detailed in Figure 2.9 (Chapter 2.3.2). The *client* sends its data to *Node* A only.

A.4.4.1 Create Backup

The *client* wants to create a backup.

1. The *client* asks the *management* for a list of *nodes*. The *management* returns a sorted list of all *nodes* (message: ***get target nodes***) (sorting might be based on specific configuration).
   - If the *management* is down and this is a first time backup, the *client* records a message and aborts.
   - If the *management* is down, *client* tries the same *nodes* as last time.
2. The *client* tries to contact *nodes* in the presorted order.
   (a) The *client* sends a backup request (message: ***get designation***).
   (b) The *node* acknowledges or denies the backup request (message: ***return designation***).
   (c) If the *node* acknowledges, it is declared as *designated node*.

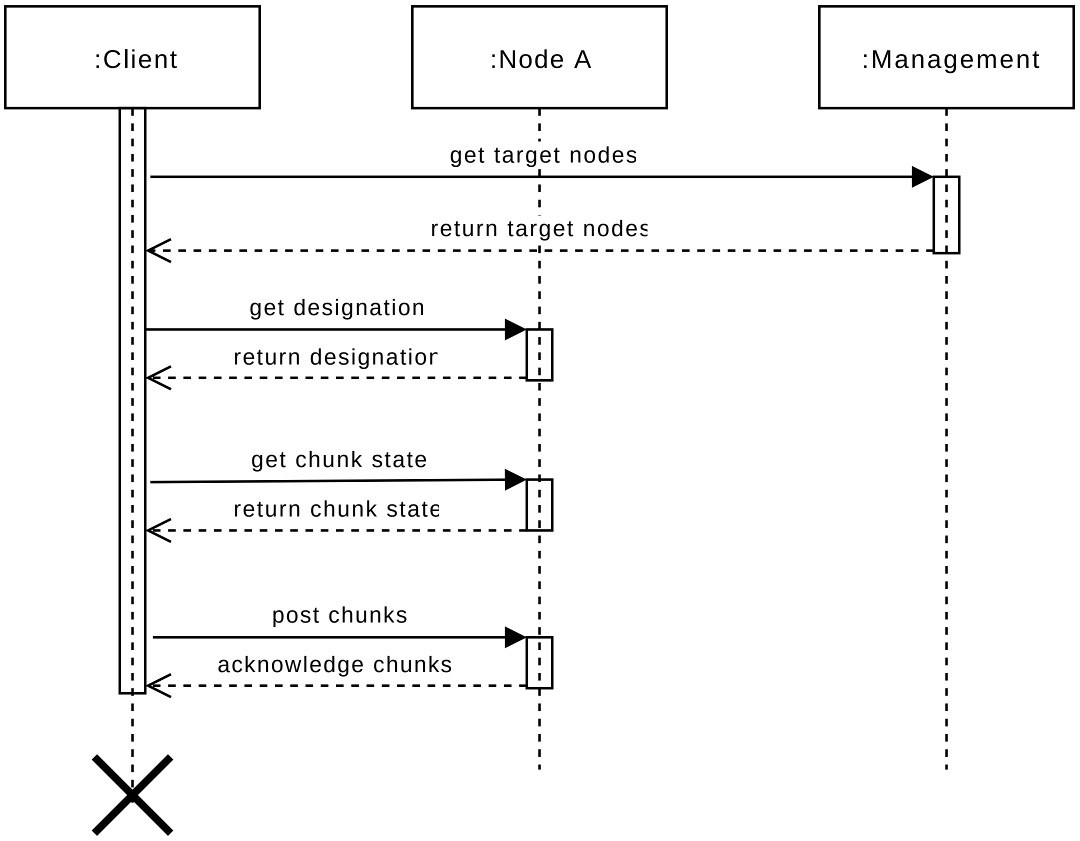

FIGURE 7: Create Backup UML Sequence Diagram

(d) If the *node* denies the backup request, the *client* tries the next *node*.

(e) If no *node* answers the request, the *client* records an error message and aborts.

3. If the backup request was successful, the *client* starts creating a backup.

(a) The *client* splits all *files* that changed since the last backup into *chunk contents* and calculate a corresponding hash, the *chunk identifier* and add it to the local *chunk index*.

(b) *Files* that have not changed since the last backup, are already be present in the *chunk index*.

(c) The *client* send all *chunk identifiers* present in the *chunk index* combined with an *expiration date* to the *designated node*. (message: ***get chunk states***)

(d) The *designated node* checks, if all *chunk identifiers* received from the *client* are present in its *chunk table*.

- If a *chunk identifier* is already present on the *designated node*, update its *expiration date* if it is further in the future.
- If a *chunk identifier* is not present on the *designated node*, request it from the *client* (see message response ***return chunk states*** for further details)

(e) The *client* sends the requested *chunk contents* to the *designated node* with a ***post chunks*** message.

i. The *designated node* verifies and persists the *chunk contents* into its *storage*. Afterwards, it acknowledges receipt to the *client* (see ***acknowledge chunks*** response message).
ii. The *designated node* replicates *chunk content* and their *expiration date* in a continuous replication process. (See scenario A.4.4.6 Data Replication)

(f) The *client* serializes its *chunk index* into a *serialised chunk index* and splits it into *chunks* as well.
(g) The *client* sends the additional *chunk contents* (as ***post chunks*** messages) to the *designated node*, in which the *root handle* is highlighted.

Special cases

- If the *client* is suspended while running, it continuous with the backup process on resume.
- If a *file* is changed during the backup process, and the *chunk contents* and *chunk identifiers* cannot be calculated, the backup process must be restarted.
- If *client* crashes, the backup is aborted and won't be continued if the *client* is restarted.
- If the *designated node* goes away (disconnects/crashes/shuts down) during the backup process, the *client* tries to resume the process. After a certain time (e.g. 15m), the *client* gives up and restarts the backup process from the beginning.
- A *node* must reject (i.e. not acknowledge) *chunk contents*, if the timestamp of the sending party (e.g. *client*) is e.g. an hour in the future or past to prevent data loss on bad synchronized clocks.
- If the *designated node* runs out of storage capacity, it does reject further *chunk contents*. After a timeout, the *client* restarts the backup process (with another *designated node*).

Possible simplifications in this study project

3a) Do not split *files* into *chunks* but send them as is.

A.4.4.2 Backup Restore

The *client* wants to restore specific *files*.

1. Same as in scenario A.4.4.1 Create Backup step 1.
2. The *client* contacts the *nodes* in the presorted order
   (a) The *client* sends a ***get root handles*** requests to a *node*, the *designated node*.
   (b) The *designated node* returns all *root handles* present in the *system* (see ***return root handles*** message)
   (c) If no *node* answers the request,the *client* records an error message and aborts.
3. The *client* fetches all *chunk contents* of the *serialised chunk indexs* from the *designated node* (message: ***get chunks***) and reassembles the *chunk indexs*.

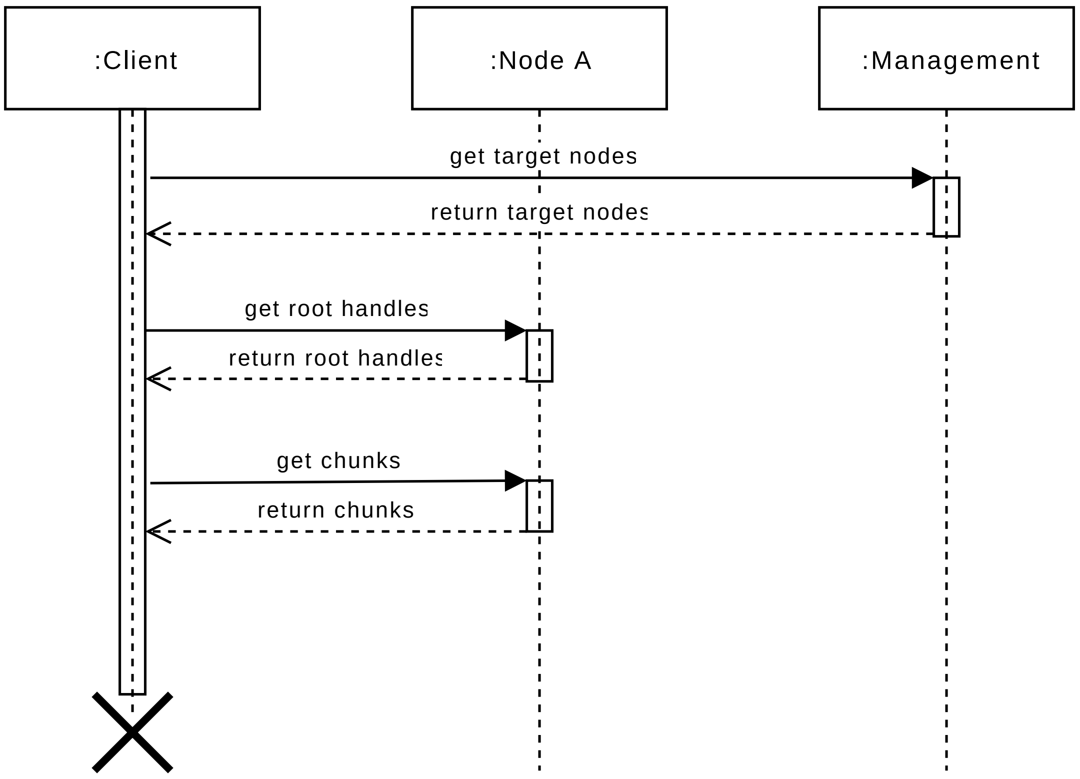

FIGURE 8: Backup Restore UML Sequence Diagram

4. The *user* specifies which *files* from which *chunk index* shall be restored.
5. The *client* looks up the *chunk identifiers* in the corresponding *chunk index*.
6. The *client* requests the *chunk contents* from the *designated node*. (message: ***get chunks***)
7. The *client* reassembles the *chunk contents* into *files*.

Special cases

- If the *designated node* does not have a requested *chunk identifier* in its *chunk table*, it requests the corresponding *chunk content* recursively.
- If the *client* crashes, the restore process must be repeated
- If the *designated node* is unavailable, the *client* selects a new *designated node* after a certain timeout (e.g. 5m)

Possible simplifications in this study project

- The *client* must request a *node* that has the data already available (possibly the *designated node*, where the backup was created).

A.4.4.3 Node Joining

A *new node* joins the *system*.

This scenario is described as it is implemented in the study project. It might be subject to further evaluation in the future.

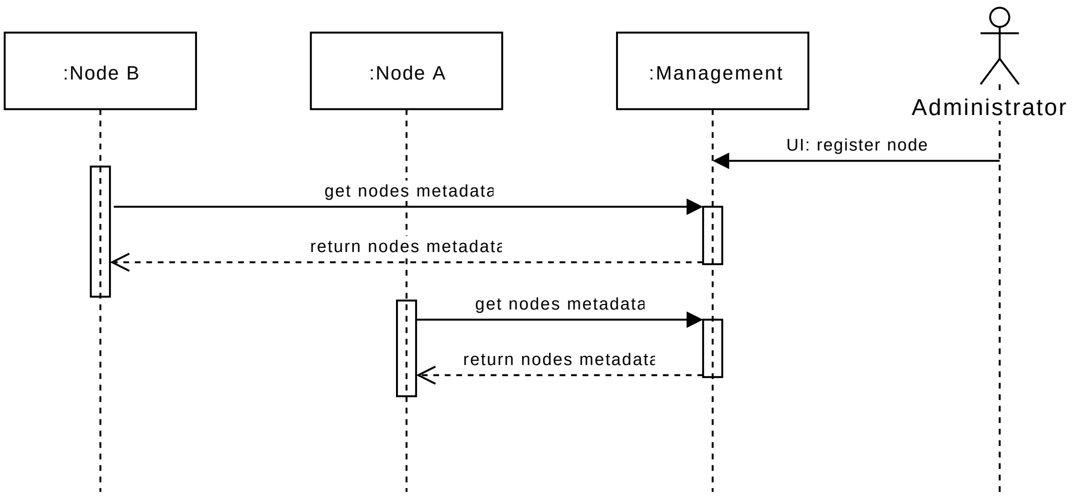

FIGURE 9: Node Joining UML Sequence Diagram

1. The *new node* is registered by an *administrator* in the *management* using its *node identifier*. A *location* must also be assigned to the *node*.
2. The *new node* queries *node metadata* from the *management* (message: ***get nodes metadata***) on startup.
3. The *new node* configures itself based on its *node identifier* and received *node metadata* from *management*.
   - If the *management* has no information available (yet) or is unavailable, the *new node* retries after a certain timeout (e.g. 5min).
4. Other *nodes* learn about the *new node* through periodical *node metadata* queries (message: ***post nodes metadata***)
   (a) If the *new node* contacts an existing *node*, before the existing *node* updates its *node metadata*, the existing *node* should query the *management* (message: ***get nodes metadata***)
   (b) If the *management* is unavailable, the *new node* ignores all communication attempts.

A.4.4.4 Node Leaving Planned

A *node* leaves the *system* planned (refered as *leaving node*).

This scenario is described as it is implemented in the study project. It might be subject to further evaluation in the future.

1. The *leaving node* is marked as ***leaving soon*** by the *administrator* in the *management*.
2. As soon as the *leaving node* realises that it is in *node state* ***leaving soon*** (using ***post nodes metadata***), it ignores messages from other *nodes*, rejects any new backups and starts replicating its *chunk contents* to another *node*.
3. As soon as all *chunk contents* are replicated onto other *nodes*, the *leaving node* changes its *node state* to ***left*** and informs the *management* (message: ***post nodes metadata***).

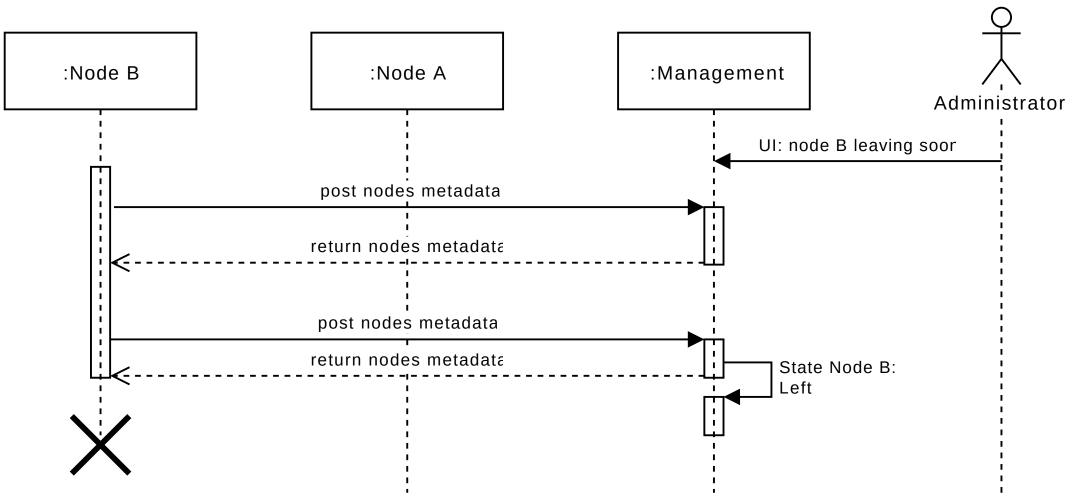

FIGURE 10: Node Leaving Planned UML Sequence Diagram

A.4.4.5 Node Leaving Unplanned

A *node* leaves the *system* unexpectedly.

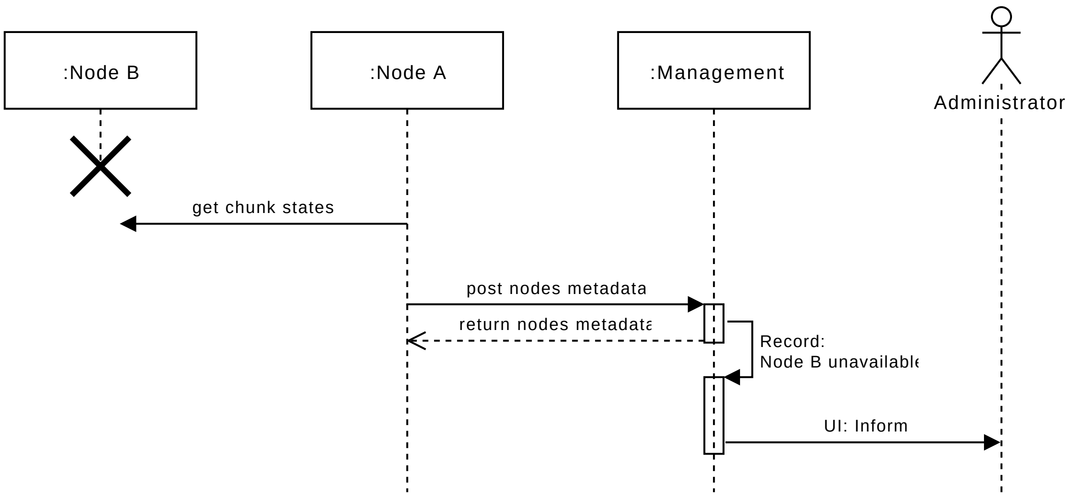

FIGURE 11: Node Leaving Unplanned UML Sequence Diagram

This scenario is described as it is implemented in the study project. It might be subject to further evaluation in the future.

1. If a *node* is not responding (which means, no other *node* nor the *management* can reach it), the *management* records the unavailability and informs the *administrator*.

- If the *node* returns, it carries on as if nothing had happened.
- If the *node* does not return, the *administrator* marks the *node* as ***left***, which is propagated to all *nodes* via the ***post nodes metadata*** message.
  - *chunk contents*, which were not previously replicated from the *node* are lost.

A.4.4.6 Data Replication

The network distributes *chunks*.

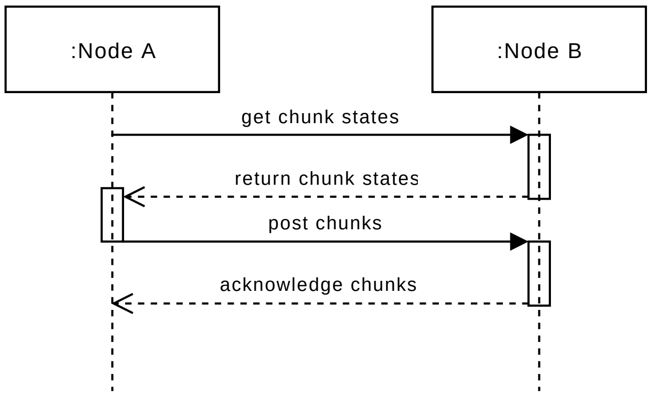

FIGURE 12: Data Replication UML Sequence Diagram

This scenario is described as it is implemented in the study project. It might be subject to further evaluation in the future.

1. The *sending node* picks *n* random entries from its *chunk table*. In the future, these entries might be selected based on heuristics.
2. The *sending node* picks one random *designated node* from its *node metadata*. In the future, the *designated node* might be selected based on heuristics.
3. The *sending node* sends the *chunk identifiers* of the chosen entries to the *designated node* with a ***get chunk states*** message.
4. The *designated node* checks, if all *chunk identifiers* received from the *sending node* are present in its *chunk table*.
   - If a *chunk identifier* is already present, update its *expiration date* if it is further in the future.
   - If a *chunk identifier* is not present, request the corresponding *chunk content* from the *sending node* (see message ***return chunk states*** for further details).
5. The *sending node* sends the requested *chunk contents* to the *designated node* with a ***post chunks*** message.
   - The *designated node* verifies and persists the *chunk contents* into its *storage*. Afterwards, it acknowledges receipt to the *sending node* (see ***acknowledge chunks*** response message).

A.4.4.7 Data has Expired Lifetime

A *node* wants to delete *chunks* with a past *expiration date*.

This scenario is described as it is implemented in the study project. It might be subject to further evaluation in the future.

- The *management* monitors the *nodes* physical timestamp and records any deviation from the *management* time larger than 1h.
- Therefore, a first prototype, the *node* may just delete *chunks* after the specified *expiration date* without further checks.

  - We build on the assumption, that the *nodes* have different upstream time-servers, so that a general time shift is unlikely.
  - In case a single *node*s time is in the past, and runs out of storage capacity, it might lead to a redundancy loss.
  - If a single *node*s time is in the future, the redundancy is reduced by one.

In the future, it is advisable to implement a consent protocol to reduce the likelihood of data loss.

A.4.4.8 Data Storage has Errors

The *storage* component notifies its *node* that a *chunk* stored in it is corrupt.

This scenario is described as it is implemented in the study project. It might be subject to further evaluation in the future.

1. The *storage* informs the *node* of a corrupt *chunk* using its *chunk identifier*
2. The *node* removes the *chunk identifier* from the *chunk table*
   - The *chunk* will be replicated eventually.
3. Inform *management* when sending the next ***post nodes metadata*** message.

A.4.4.9 Management Problems

- The case that the *management* has outdated *node metadata* due to data loss will be ignored during this study project.

A.4.4.10 Network Availability Problems

Behaviour in case a node can reach only some nodes directly The still available *nodes* should try to fulfil redundancy needs as good as possible. If a *node* cannot reach other *nodes*, it notifies the *management* with the next ***post nodes metadata***

Behaviour in case the network gets partitioned (, and the *nodes* in each partition can only reach each other and not the other ones.)

See first case in scenario A.4.4.10 Network Availability Problems

Behaviour in case nodes have different states of management information The redundancy is temporary reduced to the *nodes* which the group of old *nodes* know, but will eventually be resolved as *nodes* fetch *node metadata*.

A.4.5 Messages

Messages denote the communication units between application components. The following message specification is not bound to a specific message format (e.g. BSON or Protobuf) on purpose since it might change in the future to optimise performance.

A *message* is separated into a *message header* and *message payload* section. In the future, the *message header* section might get extended by e.g. a `message size` and `checksums`.

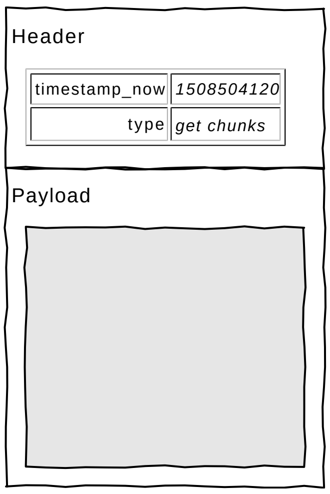

FIGURE 13: Schematic of a *message*

`header`	Set of *header fields* elements as specified in Table 2.
`payload`	The actual payload of the message depending on the message type defined in the *header field* `type` (see Table 3).

TABLE 1: Structure of a *message*.

`field name`	Unique name of the *header field*. A core set of *header field* that must be sent with every request is defined in Table 3.
`field value`	The value belonging to the field name.

TABLE 2: Structure of a *header field* Element

`timestamp_now`	The current UTC time of the local clock as a unix timestamp (64-bit).
`type`	A string representation of the message type, as described in subsections A.4.5 Messages.

TABLE 3: Core set of *header fields* sent with every Request

A.4.5.1 get target nodes

Context	A.4.4.1 Create Backup A.4.4.2 Backup Restore
Sender	A *client*
Receiver	The *management*
Additional message headers	No additional *message headers* required
Message payload	The *message payload* is empty
Response	A.4.5.2 return target nodes (compulsory)

TABLE 4: `get target nodes` Message Specification

A.4.5.2 return target nodes

Context	Response to A.4.5.1 get target nodes
Sender	The *management*
Receiver	The *client* that sent the `get target nodes` message
Additional message headers	No additional *message headers* required
Message payload	See Table 6
Response	No response

TABLE 5: `return target nodes` message specification

`target_nodes`	A list of `target node elements` as defined in Table 7. The list is sorted in descending order by preference.

TABLE 6: Structure of the `return target nodes` *message payload*

`node_identifier`	*Node identifier* of the *node*.
`address`	Address (domain name or IP) of the *node*.
`port`	Port-Number of the *node*.

TABLE 7: Structure of the `target node element`

A.4.5.3 get designation

Context	A.4.4.1 Create Backup
Sender	A *client*
Receiver	A *node*
Additional message headers	No additional *message headers* required
Message payload	See Table 9
Response	A.4.5.4 return designation (compulsory)

TABLE 8: `get designation` message specification

`estimate_size`	Backup size estimate by the *client* in bytes.
`expiration_date`	Unix timestamp (64-bit) of UTC timezone, until when the backup is scheduled to be kept.

TABLE 9: Structure of the ***get designation*** *message payload*.

A.4.5.4 return designation

Context	Response to A.4.5.3 get designation
Sender	The *node* that received the `get designation` message
Receiver	The *client* that sent the `get designation` message
Additional message headers	No additional *message headers* required
Message payload	See Table 11
Response	No response

TABLE 10: `return designation` message specification

`designation`	Boolean, whether or not the node is ready to receive the backup.

TABLE 11: Structure of the ***return designation*** *message payload*.

A.4.5.5 get chunk states

Context	A.4.4.1 Create Backup A.4.4.6 Data Replication
Sender	A *client* or a *node*
Receiver	A *node*
Additional message headers	No additional *message headers* required
Message payload	See Table 13
Response	A.4.5.6 return chunk states (compulsory)

TABLE 12: `get chunk states` message specification

`chunks`	A set of `chunk elements` as defined in Table 14 for which the state shall be checked.

TABLE 13: Structure of the `get chunk states` *message payload*

`chunk_identifier`	The *chunk identifier* of this *chunk*
`expiration_date`	The *expiration date* of this *chunk*
`root_handle`	Boolean, whether this *chunk* is a *root handle*

TABLE 14: Structure of the `chunk element`

A.4.5.6 return chunk states

Context	Response to A.4.5.5 get chunk states
Sender	The *node* that received the `get chunk states` message
Receiver	The *client* or *node* that sent the `get chunk states` message
Additional message headers	No additional *message headers* required
Message payload	See Table 16
Response	No response

TABLE 15: `return chunk states` message specification

`chunks`	A set of `chunk elements` as defined in Table 14 that are present on this node with the updated `expiration_date`

TABLE 16: Structure of the `return chunk states` *message payload*

A.4.5.7 post chunks

Context	A.4.4.1 Create Backup A.4.4.6 Data Replication
Sender	A *client* or a *node*
Receiver	A *node*
Additional message headers	No additional *message headers* required
Message payload	See Table 18
Response	A.4.5.8 acknowledge chunks (compulsory)

TABLE 17: `post chunks` message specification

`chunks`	A set of `chunk content elements` as defined in Table 19.

TABLE 18: Structure of the `post chunks` *message payload*

`chunk_identifier`	The *chunk identifier* of this *chunk*
`expiration_date`	The *expiration date* of this *chunk*
`root_handle`	Boolean, whether this *chunk* is a *root handle*
`chunk_content`	The binary *chunk contents* of this *chunk*

TABLE 19: Structure of the `chunk content element`

A.4.5.8 acknowledge chunks

Context	Response to A.4.5.7 post chunks
Sender	The *node* that received the `post chunks` message
Receiver	The *client* or *node* that sent the `post chunks` message
Additional message headers	No additional *message headers* required
Message payload	See Table 21
Response	No response

TABLE 20: `acknowledge chunks` message specification

`chunks`	A set of `chunk elements` as defined in Table 14 that were received and stored.

TABLE 21: Structure of the `acknowledge chunks` *message payload*

A.4.5.9 get chunks

Context	A.4.4.2 Backup Restore
Sender	A *client*
Receiver	A *node*
Additional message headers	No additional *message headers* required
Message payload	See Table 23
Response	A.4.5.10 return chunks (compulsory)

TABLE 22: `get chunks` message specification

`chunk_identifiers`	A set of *chunk identifiers* for which the *chunk contents* shall be returned.

TABLE 23: Structure of the `get chunks` *message payload*

A.4.5.10 return chunks

Context	Response to A.4.5.9 get chunks
Sender	The *node* that received the `get chunks` message
Receiver	The *client* that sent the `get chunks` message
Additional message headers	No additional *message headers* required
Message payload	See Table 25
Response	No response

TABLE 24: `return chunks` message specification

`chunks`	A set of `chunk content elements` as defined in Table 19.

TABLE 25: Structure of the `return chunks` *message payload*

A.4.5.11 get root handles

Context	A.4.4.2 Backup Restore
Sender	A *client*
Receiver	A *node*
Additional message headers	No additional *message headers* required
Message payload	The *message payload* is empty.
Response	A.4.5.12 return root handles (compulsory)

TABLE 26: `get root handles` message specification

A.4.5.12 return root handles

Context	Response to A.4.5.11 get root handles
Sender	The *node* that received the `get root handles` message
Receiver	The *client* that sent the `get root handles` message
Additional message headers	No additional *message headers* required
Message payload	See Table 28
Response	No response

TABLE 27: `return root handles` message specification

`root_handle_chunks`	A set of `chunk content elements` as defined in Table 19, all of which are *root handles*

TABLE 28: Structure of the `return root handles` *message payload*

A.4.5.13 get nodes metadata

Context	A.4.4.3 Node Joining Periodical *node* updates: *node* reads data from *management* only
Sender	A *node*
Receiver	The *management*
Additional message headers	No additional *message headers* required
Message payload	The *message payload* is empty.
Response	A.4.5.15 return nodes metadata (compulsory)

TABLE 29: `get nodes metadata` message specification

A.4.5.14 post nodes metadata

Context	A.4.4.3 Node Joining Periodical *node* updates: Node sends data
Sender	A *node*
Receiver	The *management*
Additional message headers	No additional *message headers* required
Message payload	See Table 31
Response	A.4.5.15 return nodes metadata (compulsory)

TABLE 30: `post nodes metadata` message specification

`this_node`	An `internal node metadata element` as described in Table 32 describing the sending *node*.
`other_nodes`	A set of `node state metadata elements` as described in Table 33. One element for for every other *node* in the *system* about which the sending *node* is aware of.

TABLE 31: Structure of the `post nodes metadata` *message payload*

`node_identifier`	The *node identifier* of this *node*
`state`	The inner *node state* of this *node*
`storage_used`	Free *storage* of this *node* in ***bytes*** .
`storage_total`	Total *storage* capacity of this *node* in ***bytes***.

TABLE 32: Structure of the `internal node metadata element`

`node_identifier`	The *node identifier* of this *node*
`state`	The *node state* of this *node* (as seen from the outside)

TABLE 33: Structure of the `node state metadata element`

A.4.5.15 return nodes metadata

Context	Response to A.4.5.14 post nodes metadata Response to A.4.5.13 get nodes metadata
Sender	The *management*
Receiver	The *node* that sent the `post nodes metadata` / `get nodes metadata` message
Additional message headers	No additional *message headers* required
Message payload	See Table 35
Response	No response

TABLE 34: `return nodes metadata` message specification

`nodes`	A set of `node contact metadata elements` as described in Table 36. One element for for every *node* in the *system*.

TABLE 35: Structure of the `return nodes metadata` *message payload*

`node_identifier`	The *node identifier* of this *node*.
`address`	The address of this *node*.
`port`	The *nodes* port of this *node*.
`state`	The *node state* of this *node*.

TABLE 36: Structure of the `node contact metadata element`

A.5 Language Evaluation

Based on the Requirements described in Appendix A.3, we compared three languages/technologies.

Rust is a modern language representing an alternative to the traditional "low-level" languages C and C++.

Erlang was our second option since it is well suited for distributed, fault-tolerant systems. It is also a functional programming language, with which we both do not have much experience but share an interest in.

Go came up when we were researching for distributed systems in Rust. Friends, as well as various blog posts, suggested that Go is established in the field of distributed systems, has a diverse eco-system and is therefore well suited for the job.

We grouped the criteria for our comparison into three groups: Client (See Table 37), Distributed System (See Table 38) and Eco System (See Table 39). The assessment is not entirely objective and based on brief research as well as personal impressions.

A.5.1 Decision

We decided to use Rust. Despite its rather young age, Rust is supported by a very enthusiastic and welcoming community and provides stable releases and an excellent language documentation.

The main reason we settled for Rust was its type system as well as the concept of Ownership and Borrowing, which enables so many of Rusts features and permits us to do safe concurrency. Also, Rust has two excellent networking libraries.

The most significant downside of Rust for us is the limited amount of high-quality libraries.

We ruled out Erlang because of three reasons. Firstly, since we do not have any experience with functional programming languages, we expect a much steeper learning curve and therefore a slower progress. Secondly, we think that we do not need most of Erlangs key features such as Hot Swapping and strong process isolation. At last, Erlang requires to be run in the Erlang VM, which makes deployment more difficult than in Rust or Go.

In comparison to Rust, Go lacks various features. For example, the lack of generics, a need for garbage collection and pointers lead to more memory consumption and higher parallelisation risks. An advantage of Go is a slightly more established ecosystem.

TABLE 37: Language and Ecosystem Comparison for the Client

Language / Criteria	Rust	Erlag	Go
platform independent	Yes, LLVM Backend [16]	Yes [25]	Yes [29]
allow an easy installation (no runtime required or bundle it)	Runtime included, compiles to machine code via LLVM [16]	No [18]	Creation of statically-linked binaries by default [31]
bindings to a reasonable (platform independent) UI-Framework	No, lots of tools are in in alpha stage. See `https://github.com/rust-unofficial/awesome-rust#gui`	No [6]	Yes, See `https://github.com/avelino/awesome-go#gui`

TABLE 38: Language and Ecosystem Comparison for the Distributed System

Language / Criteria	Rust	Erlag	Go
run on lightweight (cost effective) platforms such as a raspberry pi.	Yes, through LLVM [16]	Yes [25]. Different Semantics: Run in a cluster	Yes [29]
fast / efficient	Yes, ”zero cost abstraction” and ”minimal runtime” [34], LLVM optimizer [16]	Yes, using the Actor Model [12]	Yes, using the Goroutines [8]
safe concurrency for networking and calculation of checksums	Yes, using borrow and move semantics [32]	Yes, using the Actor Model [12]	Yes, using the Goroutines [8]
Use existing libraries (e.g. for reading disk health status)	Yes, using FFI [33]	Yes, using NIF [24]	Yes, using cgo [27]
provide a stable and fast network stack	Yes, in the standard library as well as frameworks such as tokio [38]	Yes, in the standard library [26]	Yes, in the standard library [30]

TABLE 39: Ecosystem Comparison

Language / Criteria	Rust	Erlag	Go
Web frameworks (for the management)	Yes, e.g. `https://gotham.rs/`	Yes, e.g. `http://chicagoboss.org/`	Yes, e.g. `https://beego.me/`
Stability	"stability without stagnation" [17]	Yes, first release in 1986 [26]	Stabile, release every 6 months [28]
stable libraries for common tasks	More than 11'000 on `https://crates.io/`	More than 5'000 Packages available on `https://hex.pm/`	More than 756'00 golang Repositories indexed on `http://go-search.org/` (includes all GitHub forks as well)
productive tooling	Cargo, IDE-Support (VS-Code, IntelliJ Rust etc)	rebar3, IDE-Support (erlide, IntelliJ Erlang etc)	dep not yet that established but dependency management via 'go get' IDE-Support (VS-Code, Goland etc)
good testing frameworks (unit and integration tests)	Included (cargo test) as well as other testing frameworks	Included (EUnit)	Included (testing) as well as other testing frameworks
good and up to date documentation	Yes, Rust Book, Good Documentation of the standard library, other Literature	Yes, User Guide, Good Documentation of the standard library, other Literature	Yes, Good Documentation of the standard library, other Literature
active community to get support	Yes, via IRC, Forum and more. See `https://www.rust-lang.org/en-US/community.html`	Yes via IRC/Slack, Forum and more. See `http://www.erlang.org/community`	Yes via IRC/Slack, Forum and more. See `http://www.erlang.org/community`
Support bug-free coding (e.g. good type system, memory safety, linting or good compiler)	Yes, Linter, Borrow-Checker. No good metrics / coverage libraries yet.	Yes, Linter, Coverage Tools etc.	Yes, Linter, Coverage Tools etc.

A.6 Prototype Command Line Interface

A.6.1 Client

```
$ redbackup-client-cli --help
redbackup client-cli 0.1.0
Raphael Zimmermann <dev@raphael.li>:Fabian Hauser <fabian@fh2.ch>
redbackup client

USAGE:
    redbackup-client-cli [OPTIONS] [SUBCOMMAND]

FLAGS:
        --help       Prints help information
    -V, --version    Prints version information

OPTIONS:
        --chunk-index-storage <chunk-index-storage> Folder where chunk
            ↪ indices are stored. [default: /tmp/]
    -h, --node-hostname <node-hostname>          hostname of the node
        ↪ to contact [default: 0.0.0.0]
    -p, --node-port <node-port>                  port of the node to
        ↪ contact [default: 8080]

SUBCOMMANDS:
    create     Create a new backup
    help       Prints this message or the help of the given subcommands
    list       List available backups on the node.
    restore    List available backups on the node.
```

A.6.1.1 Create

```
$ redbackup-client-cli create --help
redbackup-client-cli-create
Create a new backup

USAGE:
    redbackup-client-cli create [OPTIONS] <expiration-date> <local-
        ↪ backup-dir>

FLAGS:
    -h, --help
            Prints help information

    -V, --version
            Prints version information


OPTIONS:
        --exclude-from <FILE>
```

```
          Exclude multiple glob patterns from FILE. Define one pattern
             ↪  per line. Patterns are relative to the backup
          root, e.g. 'pictures/**/*.jpg'. For allowed glob syntax, see
          https://docs.rs/glob/0/glob/struct.Pattern.html#main

ARGS:
    <expiration-date>
          the expiration date of this snapshot(format: %Y-%m-%dT%H:%M)

    <local-backup-dir>
          Directories, that should be backuped
```

A.6.1.2 List

```
$ redbackup-client-cli list --help
redbackup-client-cli-list
List available backups on the node.

USAGE:
    redbackup-client-cli list

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information
```

A.6.1.3 Restore

```
$ redbackup-client-cli restore --help
redbackup-client-cli-restore
List available backups on the node.

USAGE:
    redbackup-client-cli restore <backup-id> <local-restore-dir>

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information

ARGS:
    <backup-id>            ID of the backup that should be restored
    <local-restore-dir>    Destionation, where the files should be
        ↪  restored to.
```

A.6.2 Node

```
$ redbackup-node-cli --help
redbackup node-cli 0.1.0
Raphael Zimmermann <dev@raphael.li>:Fabian Hauser <fabian@fh2.ch>
```

```
redbackup node server

USAGE:
    redbackup-node-cli [OPTIONS] [--] [ARGS]

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information

OPTIONS:
    -k, --known-node <known-node>... ip address and port (<ip-address
        ↪ >:<port>) of other known nodes in the network

ARGS:
    <ip>             IP to bind [default: 0.0.0.0]
    <port>           IP to bind [default: 8080]
    <storage-dir>    path to the storage directory [default: ./data/]
    <db-file>        path to the database file [default: db.sqlite3]
```

A.7 Personal Reports

A.7.1 Raphael Zimmermann

The project was a fascinating intellectual challenge, which allowed me to use lots of my experience and knowledge acquired while studying.

Finding an appropriate way of documenting a complex software system was one of the significant challenges for me. It was interesting to attend the lecture "Application Architecture" in parallel which focuses on this problem entirely. Unfortunately, many interesting techniques were discussed relatively late in this course by which we already finished our specification. Reading about fault tolerance and networking patterns in the lecture "Advanced Patterns and Frameworks" also strengthened my confidence in our proposed architecture because we used many of them implicitly.

I played well together with Fabian even though we worked a lot apart. We found the right balance by discussing relevant tasks in person and doing more routine and detailed work separate and more focused.

The explorative nature of the project made it hard to come up with reasonable estimates. It was the right decision to keep sprints very short to stay agile. I also learned to appreciate the value of checklists, e.g. for sprint planning and completion.

A.7.2 Fabian Hauser

"It doesn't matter how beautiful your theory is...
If it doesn't agree with experiment, it's wrong." - Richard Feynman

This quote by Richard Feynman summarises pretty well, what made this project very interesting to me - building both, a high level architecture and a concrete, working prototype. During our studies, we learned a lot about tools to create and design systems, but never had the actual chance to build a larger project with them. I am very delighted for this opportunity.

Something that I particularly enjoyed was working together with Raphael. As we come from different specialised backgrounds, with Raphael coming from a software and myself more from a system engineering side, our discussions regarding the application architecture were both very valuable and enriching.

Wanting to learn Rust for a while, this project seemed like the ideal opportunity to do so. For me, this proved to be one of the main challenges during this project thought, as Rust has a very steep learning curve. This mainly had impact on the programming productivity, although this probably would have been even worse with the choice of a functional language. Still, I am glad to have had the opportunity to learn Rust.

Declaration of Authorship

We, Fabian HAUSER and Raphael ZIMMERMANN, declare that this thesis and the work presented in it are our own, original work. All the sources we consulted and cited are clearly attributed. We have acknowledged all main sources of help.

Fabian Hauser

Raphael Zimmermann

Rapperswil, December 20, 2017