FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Privacy-Preserving Storage Service for the Fog

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Michael Fabsich
Matrikelnummer 1226353

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dr.-Ing. Stefan Schulte
Mitwirkung: Vasileios Karagiannis, MSc.

Wien, 17. August 2020

Michael Fabsich	Stefan Schulte

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

A Privacy-Preserving Storage Service for the Fog

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Michael Fabsich
Registration Number 1226353

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dr.-Ing. Stefan Schulte
Assistance: Vasileios Karagiannis, MSc.

Vienna, 17th August, 2020

Michael Fabsich

Stefan Schulte

Erklärung zur Verfassung der Arbeit

Michael Fabsich

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 17. August 2020

Michael Fabsich

Acknowledgements

First of all, I want to thank my advisors Professor Stefan Schulte and Vasileios Karagiannis for their invaluable feedback and permanent support. In particular, the various discussions with Vasileios Karagiannis were essential for the success of this work.

I thank my family who supported me continuously and tirelessly during my studies. Especially, I want to thank my girlfriend Stephanie for her constant encouragement and endless patience in all these years. Finally, I want to thank all my friends and fellow students who accompanied me through the time of my study.

Kurzfassung

Die Verwendung von Diensten zur cloud-basierten Datenspeicherung wird immer populärer. Diese Dienste bieten Alternativen zu traditionellen lokalen Speicher-Lösungen an und haben viele Vorteile. Unter anderem können die gespeicherten Daten von überall dort abgerufen werden, wo Internet-Zugriff vorhanden ist. Außerdem können die Kosten für die IT Infrastruktur gesenkt werden. Trotzdem gibt es einige Nachteile bei der Verwendung von cloud-basierten Diensten zur Datenspeicherung. Diese Nachteile beziehen sich insbesondere auf den Datenschutz, die Latenz und die Verfügbarkeit der Daten. Die Nutzer müssen dem Anbieter und den verwendeten Sicherheitsmechanismen vertrauen. Außerdem werden die Daten in zentralen Rechenzentren gespeichert, die normalerweise nicht in der Nähe der Nutzer sind. Dadurch erhöht sich die Latenz beim Zugriff auf die Daten. Zusätzlich ist der Nutzer davon abhängig, wie der Anbieter des Cloud-Dienstes diesen Dienst betreibt. Es könnte sein, dass der Dienst temporär nicht verfügbar ist oder, dass sich der Anbieter sogar dafür entscheidet, den Dienst abzuschalten. Dadurch könnte der Zugriff auf die Daten verloren gehen.

Das Ziel dieser Arbeit ist es, ein Speichersystem zu designen und zu implementieren, welches diese Nachteile der Cloud-Dienste ausmerzt. Unser Ansatz basiert auf Fog Computing, was erlaubt, Ressourcen am Rande des Netzwerks nutzen zu können. Wir zielen darauf ab, die Daten vor unbefugtem Zugriff zu schützen, die Latenz des Up- und Downloads zu reduzieren und die Daten hochverfügbar bereitzustellen. Dazu entwerfen wir in dieser Arbeit ein Speichersystem, das cloud-basierte Dienste zur Datenspeicherung mit dem Ansatz des Fog Computings kombiniert. Ein wesentlicher Bestandteil des Systems ist die "Placement Strategy", mit der die Daten auf verschiedene Komponenten verteilt werden. Wir verwenden verschiedene etablierte Mechanismen und Techniken, um die Ziele zu erreichen. Dazu gehört unter anderem Erasure Coding, aber auch die Verwendung von Kryptographie.

Wir vergleichen unsere Lösung mit anderen Ansätzen im Zusammenhang mit cloud-basierter Datenspeicherung und Lösungen, die auf Fog Computing zurückgreifen. Basierend auf Analysen und Experimenten bewerten wir unseren Ansatz und zeigen, dass das implementierte System die Daten schützt, eine geringe Latenz für Up- und Downloads bietet und eine hohe Verfügbarkeit der Daten sicherstellt. Insbesondere zeigen wir, dass unser implementiertes System die Latenz beim Upload um 15-42% und beim Download sogar um 58-76% im Vergleich zu einer reinen Cloud-Lösung reduzieren kann.

Abstract

Nowadays, cloud storage services have become very popular and are used by both organizations and individuals. These services offer various advantages for their users compared to traditional local storage systems. For example, the users have ubiquitous access to these services and their stored data via the Internet. Besides that, the IT maintenance costs can be lowered as no local storage system has to be set up and maintained. Nevertheless, cloud storage services have some limitations. Specifically, these limitations are related to privacy, latency and availability. As the cloud storage provider stores and administers the uploaded data, the users have to trust the provider of the cloud storage service and the implemented security mechanisms. Furthermore, the data is stored in central data centers which are usually not in the vicinity of the user and therefore the latency of retrieving data is increased by the geographical distance. Additionally, the user is dependent on the cloud storage provider and its handling of the cloud storage service. If the cloud storage service is not accessible temporarily or – even worse – permanently, the data is lost as the data can not be accessed or migrated.

In this thesis, we design and implement a fog-based storage system which tackles these limitations of cloud storage services. Our approach is leveraging on fog computing which enables us to use resources at the edge of the network. The main goals of our approach are preserving privacy, providing low latency, and offering high availability. To this end, we design and implement a storage system which combines cloud storage services with the approach of fog computing. An essential part of the system is our Placement Strategy which distributes the data to different storage components. We use different established mechanisms and techniques to reach the goals of our approach, e.g., erasure coding and well-known methodology in the area of cryptography.

We compare our solution with other approaches related to cloud-based and fog-based storage. Based on analyses and experiments with different scenarios, we evaluate our approach and show that the designed and implemented fog-based storage system preserves privacy, provides low latency, and offers high availability. Most notably, we show that our implemented fog-based storage system is able to reduce the latency by 15-42% in Upload Mode and even by 58-76% in Download Mode compared to a cloud-only solution.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Nowadays, cloud storage services, like Dropbox[1] or Google Drive[2], have become a popular alternative to local storage systems [1]. Currently used by both organizations and individuals, these services provide various advantages for the users. For example, after uploading data to the cloud, the files become accessible from any computer, smartphone or tablet that has access to the Internet [2]. Moreover, the users have access to the data without setting up a local storage system and do not have to be concerned with backup and preservation strategies. Furthermore, not using a local storage system can lower the IT maintenance costs [1].

Cloud storage provides access to compute and storage resources located in data centers [3]. Data centers combine all the available resources into a unified shared pool, which leads to higher utilization of the hardware and software resources and energy efficiency [4]. Despite the advantages, cloud computing still has limitations, some of which are inherited by cloud storage. Specifically, these limitations are:

1. **Privacy:** First, the data – including sensitive and critical information – is given to a cloud storage provider which has full access to the stored data. This requires that the users trust the cloud storage providers. Moreover, although the user accesses the storage through a server which protects the data with particular security mechanisms, these publicly available servers can be targeted by malicious external attacks which aim at stealing the data [2]. History shows that big service providers are a popular target for such attacks [5].

[1] https://www.dropbox.com
[2] https://www.google.com/drive/

2. **Latency:** Since the data is stored in huge compute and storage centers, a multitude of requests towards these centers can lead to a network bottleneck, which results in high latency [6]. Moreover, these centers are not in the vicinity of the user, therefore the latency of retrieving data is further increased by the long geographical distances [7].

3. **Availability:** Another limitation of a cloud storage is that the user is dependent on this particular provider. If the storage service is not reachable temporarily or even worse, the provider shuts down the cloud storage service, the data cannot be accessed or migrated to another vendor. In the literature, this is also referred to as the *vendor lock-in* risk [8].

While these limitations still apply to cloud storage, there is an emerging computing paradigm that aims at overcoming such problems in the context of the IoT (Internet of Things) [9]. This paradigm is known as *fog computing* [10]. Fog computing aims at extending cloud computing to offer services hosted on devices throughout the network and on devices located at the edge of the network [11]. These devices can be used for distributing computation, communication and storage closer to the end user [12]. Motivated by the limitations of cloud storage services related to privacy, availability and latency, this thesis will focus on leveraging the resources at the edge of the network according to the fog computing paradigm in order to tackle these limitations.

1.2 Aim of the Work

The aim of this thesis is to design and implement a storage system which tackles the limitations of traditional cloud storage by leveraging on fog computing. To reach this goal, the literature on cloud storage and fog computing will be studied and ways to combine these two fields will be explored.

The expected outcome can be roughly divided into three parts:

Placement Strategy A core part of this thesis is the design of a Placement Strategy for the data which aims at preserving privacy, providing low latency, and offering high availability.

- Privacy refers to ensuring that the stored data cannot be analyzed or read by anyone else than the user, i.e., the owner of the data [2]. Since privacy is still an open issue in traditional cloud storage systems, the approach in this thesis aims at providing a storage solution that preserves the privacy of the data.

- Latency describes the elapsed time between a request and the corresponding response citeDBLP:conf/soca/MattW017. An aim of this thesis is to reduce the overall latency of the storage system compared to a cloud-only storage service.

- Availability refers to accessing the storage whenever data is requested [2]. As cloud storage services can suffer from temporarily unavailable servers which makes the data unavailable to the user, the data placement strategy in this thesis aims at increasing availability.
- Another aim is to design the data placement strategy in a way that privacy, availability and low latency will be achieved at the same time without a trade-off among them.

Storage system The main outcome of this thesis will be a prototypical implementation of a fog-based storage system which integrates the previously defined data Placement Strategy. This implementation consists of two applications: A frontend application for the user and a middleware service which will be executed on computational resources in the fog. The frontend application will offer the possibility to up- and download files. If the user makes a request to download a file, the system gets the required data and regenerates the original file. The source code and the deployment descriptions of both parts will be part of the outcome.

Evaluation It is an aim of this thesis to provide an evaluation which verifies that there are benefits in leveraging the resources at the edge of the network and that the implemented fog-based storage system solves the problems discussed in the motivation.

1.3 Methodology and Approach

The methodological approach can be grouped roughly into four parts which can be summarized as follows:

Analysis of Related Work The area of cloud computing is already well-studied and there are different approaches that aim at tackling various related limitations. The area of fog computing has gained attention in research in the recent years. The thesis will contain an introduction to cloud and fog computing including a comparison of these two paradigms. Besides that, the existing approaches to storage systems in cloud computing and fog computing will be investigated and compared to each other.

Design This part includes the design of the data placement strategy and the architecture for the storage system. Based on the related work, appropriate mechanisms to store the data will be designed. The strategy should be able to achieve the aims related to data privacy, availability and low latency by using available resources at the edge of the network according to the fog computing paradigm. The main characteristics of the strategy can be described as follows:

- The data will be splitted by using erasure coding [13] and the resulting parts of this data will be distributed to cloud storage providers and fog devices in a way that no device will have enough information to regenerate the original data. Therefore, the data placement strategy preserves privacy.

- To reduce latency, we can distribute data to resources at the edge of the network, hence in the vicinity of the user. This reduces the overall latency of storing and retrieving operations compared to a cloud-only storage.
- The data will be distributed to multiple devices and cloud providers redundantly in order to increase availability. Even if a server is unavailable, the data would still be accessible.

Implementation The implementation of the fog-based storage system integrates the previously designed data placement strategy. The functionality of this system covers the basic features of a common storage system:

- **Upload:** The user uploads a file to the storage system using the given frontend application. The system divides the file using erasure coding and distributes the parts of the data to the cloud storage providers and devices in the fog.
- **Download:** The user requests a particular file by utilizing the frontend application. The system retrieves the needed parts from the relevant sources and regenerates the original file which is then returned to the user.

The implementation of the fog-based storage system will be realized using the following tools. For the backend service, the Java Spring Framework is used. For the frontend application, a desktop application using JavaFx is used. In order to split the files using erasure coding, the open-source Java implementation of Reed-Solomon code from Backblaze[3] is used.

Evaluation The implemented storage system is evaluated on a test bed based on cloud services and virtual machines on the edge of the network. The implementation of the fog-based storage system will allow to simulate other approaches from the literature with similar goals. All implemented approaches will be executed for random scenarios multiple times in order to provide comparisons between our proposed approach and the related approaches of the literature.

The comparisons will focus on latency by measuring end-to-end Round Trip Time (RTT) which is the time between sending a request and receiving a response [14]. Moreover, the evaluation will contain an analysis related to privacy and availability of our proposed approach and a comparison with other state-of-the-art approaches.

[3]https://www.backblaze.com/blog/reed-solomon/

1.4 Structure

The remainder of this work is structured as follows.

In Chapter 2, the needed background information for understanding the underlying concepts and technologies of our approach is presented. We discuss the concepts of Internet of Things, cloud computing and fog computing. Besides that, we describe erasure coding which is an essential technique of our approach.

Chapter 3 contains a discussion of the most relevant scientific work related to storage in the cloud and fog. We discuss the features of these approaches and compare these approaches to our approach.

In Chapter 4, we discuss the requirements for a fog-based storage system with the main goals of preserving privacy, providing low latency, and offering high availability. Besides that, we present the design and concept of our fog-based storage system which fulfills these requirements.

Based on the designed storage system, we present the details of our implementation in Chapter 5.

Chapter 6 covers the evaluation of our approach. Based on experiments and analyses, our implemented storage is verified against the requirements of Chapter 4 and is compared with other approaches from the literature.

In Chapter 7, we summarize the work done in this thesis and present an outlook on future work.

CHAPTER 2

Background

In this chapter, we discuss fundamental background information of the later referenced concepts and mechanisms. As we have mentioned in Chapter 1, the aim of this work is to design and implement a fog-based storage system. First, we describe the concept of cloud computing and the most important characteristics of cloud computing. After that, we discuss the concept of *Internet of Things* (IoT) and the integration of IoT with cloud computing. Furthermore, we show how IoT and cloud computing can benefit from each other. Afterwards, we discuss the concept of fog computing and its aims. We conclude this chapter with a description of *Erasure Coding* which is an essential technique of our approach.

2.1 Cloud Computing

Cloud computing allows to gain access to IT infrastructure, platforms, and various applications in the form of a service on the Internet. Cloud providers virtualize the physical computational infrastructure and therefore can offer cloud users an on-demand access to the desired resources which can be provisioned at almost any time [15]. By using virtualization, the cloud provider is able to allow an abstract view on a shared pool of physical resources, like servers, data storage, networks, or applications. Requests of cloud users can then be served from this pool. This can happen almost immediately as the user does not get access to a real physical machine but a virtual machine. Cloud users get access to the required resources very fast and are not in the need of establishing an own infrastructure for developing and deploying new applications. Furthermore, cloud services can be scaled dynamically. If in need, more resources can be provisioned at any time or released if the demand is shrinking again. This process can be done automatically and without requiring human interaction with the service provider [3]. Besides that, the cloud user only pays for resources which the user needs, unused resources will not be charged.

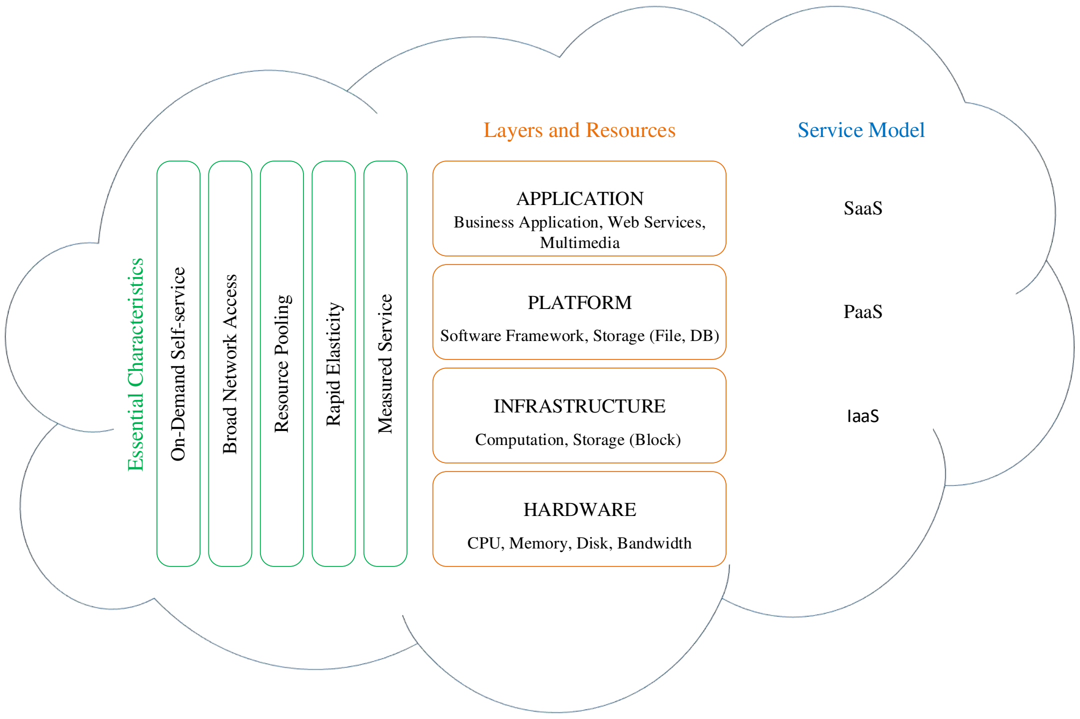


Figure 2.1: Cloud computing: overall view (from [4])

Because of its flexibility regarding resource provisioning, the use of cloud computing can reduce costs, which led to a new way of developing and deploying applications. These advantages yield to a wide use of cloud computing and therefore cloud computing has an huge impact on the IT industry [4]. Figure 2.1 depicts an overall overview of cloud computing and its layered architecture, service models and characteristics. In the following we discuss the layered architecture and the service models in detail. After that, we present the depicted essential characteristics.

2.1.1 Layered Architecture & Service Models

As shown in Figure 2.1, the architecture of cloud computing can be described using four layers: hardware, infrastructure, platform and application [4]). Every layer offers its resources as a service to the layer above and consumes the services of the layer below. The services which are offered to the cloud user are grouped into three main categories which are referred to as service models. The U.S. National Institute of Standard and Technologies (NIST) defines the following service models for cloud computing [3]:

Software as a Service (SaaS) The consumer can access a software which is running on a cloud infrastructure by using a Web interface or a programming interface. The control over the infrastructure is not in the responsibility of the consumer. Apart from limited application settings, the customer has no possibilities to configure the system.

Platform as a Service (PaaS) The consumer is able to deploy applications to the cloud environment but does not control the underlying cloud infrastructure like networks, servers operating systems, or storage. The deployed application is under the customers control and can be adjusted in every desired way.

Infrastructure as a Service (IaaS) IaaS is the most flexible offer to the customer. The consumer is able to provision processing, storage, networks, and other basic computing resources and has control over the operating system and deployed applications.

2.1.2 Characteristics

Figure 2.1 shows the essential characteristics of cloud computing, which are also defined by the NIST. These essential characteristics are:

On-demand self-service Computing capabilities can be provisioned at nearly any time automatically without involving actions of others.

Broad network access The resources of the cloud can be accessed with every device which is connected to the Internet.

Resource pooling The customer is able to gain access to the resources dynamically. The provider offers a pool of resources which are then assigned and reassigned to the need of the cloud users.

Rapid elasticity Resources and capabilities can be provisioned and released often automatically which aims at easy scaling and enables to fit the demands of the consumer. The resources are apparently available in an unlimited amount.

Measured Service The resource usage is monitored and controlled. This is essential for both the cloud user and the cloud provider, as the billing of the resources is often based on pay-per-use.

2.1.3 Deployment Models

The cloud computing model with its characteristics, layers, and service models can be deployed in different ways which are referred to as deployment models. Mainly, the deployment model is definded by who can use the services of the cloud. The NIST describes the following deployment models [3]:

Private Cloud The cloud infrastructure is only available for a single organization. Therefore, only this organization and the members of the organization can provision and access resources of this cloud. The infrastructure may be managed by this particular organization but can be offered by a third party which operates the cloud on- or off-premise.

Community Cloud The cloud infrastructure is available for a community of consumers of organizations which have shared concerns. Therefore, only these consumers can provision resources of the cloud. Similarly to the private cloud, the cloud can be managed by one of the organizations of the community or a third party.

Public Cloud The cloud infrastructure is publicly available. Resources can be provisioned by everyone. The infrastructure is managed by one organization and exists on the premises of this organization.

Hybrid Cloud The Hybrid Cloud is any combination of two or more cloud deployment models (private, community, or public).

Based on the the different deployment models and service models, the cloud providers can offer various services to the cloud users. The storage of data in the cloud is one of these services. As the cloud storage services are an essential part of our approach, we will present some details of cloud storage services as well as the benefits and drawbacks of these services in the next section.

2.1.4 Cloud Storage Services

In 2018, 68% of enterprises in the EU which are using cloud computing services, used cloud storage services to store data in the cloud [16]. Only the e-mail service was used more often by enterprises. This shows how pervasive the use of cloud storage is already. We are going to investigate the benefits and drawbacks of using cloud storage in the following.

As has already been mentioned, cloud computing services offer many advantages, hence many of them are inherited by cloud storage services. Some of them are [17]:

Cost Reduction Because of the pay-per-use billing method of the cloud providers, cloud users can make cost savings. Cloud users, such as enterprises or organizations do not have to establish an own storage infrastructure but request the storage capacity from the cloud storage service.

Convenience Besides the cost reduction, cloud users can acquire access to cloud storage via the Internet. As on-premise infrastructure of enterprises are often not accessible from outside, employees are possibly not able to access data while working remotely. Cloud storage is accessible from everywhere and with every device which has access to the Internet.

Reliability Cloud storage providers ensure the availability of the data by storing data redundantly and providing data recovery. Furthermore, the cloud providers offer a particular level of reliability which is guaranteed by their SLAs (Service Level Agreements).

Besides these advantages, the storage of data in the cloud has also some drawbacks:

Security & Privacy The data is handed over by the cloud user to the cloud storage provider and therefore the cloud user has to trust the provider to a certain degree. Due to legal rules and contracts, the cloud storage provider is not allowed to access the stored data of the cloud user, but theoretically the content of transferred data can be read and used by the provider and third parties which have access to the data. Besides that, the used hardware which is offered by the cloud providers is shared with other cloud users. The cloud storage provider has to ensure that cloud users have only access to that data which they are authorized to [2].

Vendor Lock-In As the data is stored on the infrastructure of the cloud provider, the data can not be accessed if the cloud storage service is temporarily not available. Even worse, the cloud provider could shut down the business and the data would be constantly lost [8].

Data Bottleneck The data stored by using cloud storage services is placed on storage in central data centers of the cloud storage provider. Cloud users and their applications are possibly distributed over a geographically wide area. This results in a potentially long distance between the data producer or data consumer and the data storage which leads to high latency [6].

2.2 Internet of Things

The Internet of Things (IoT) refers to a network of interconnected smart *things* and objects which communicate through the *Internet* [4]. These things – acting as sensors or actuators – in the IoT are not only complex electronic devices like mobile phones but also everyday objects like clothing, furniture, monuments, etc. [18]. Therefore, IoT applications have the potential of impacting different aspects of the every-day life [19]. IoT applications are already visible in the modern society including smart cities, smart factories, e-health, assisted living, etc. [20].

In general, the total number of smart things in use will reach 25 billion by 2021 according to Gartner [21]. Solely in the enterprise and automotive area, Gartner forecasts 5.8 billion smart things which means an increase of 21% compared to the year 2019 [22]. Cisco forecasts that IoT connections of related applications will take up to 51% of the total Internet connections in 2022 [23]. This huge number of things and the resulting data fosters the development of further new applications in different areas. This shows how pervasive the IoT will become. Based on the potential of the IoT, both industrial and research organizations have shown a lot of interest in IoT-related applications [24].

An example for an IoT application is an intelligent traffic light control. Usually, traffic lights are controlled by a pre-defined schedule which is inefficient as the traffic situation is constantly changing [25]. As the things of the IoT are able to real-time measure and process traffic data, it is possible to improve the conditions for traffic participants and the environment related to travel time, fuel consumption, pollution, and accidents [26].

Figure 2.2 shows a system architecture for intelligent traffic light control [27]. The participating components in this example are four sensors which are distributed on both sides of the road and are represented by blue dots, two vehicles which are illustrated by yellow rectangles, and the traffic light controller. The orange dashed lines represent the communication of the participating components. The vehicles calculate parameter of themselves and report the information to the sensors. The sensors collect this information from the vehicles, count the vehicles around themselves, and transfer the collected information to the traffic light controller. Based on the received information, the traffic light controller is able to adapt its schedule and to optimize the traffic flow. This example shows the potential of interconnected objects for intelligent applications.

As the objects – sensors, vehicles and traffic lights control – are not connected to the Internet and are communicating wirelessly, this system architecture is referred to as Wireless Sensor Network (WSN). The IoT uses the Internet to ensure that every object or thing has an identification, WSNs could use the Internet but the usage is not mandatory [28]. As this architecture does not involve the Internet, the schedule of the traffic light is based only on local events. However, a city-wide traffic signal control application should also consider traffic data from other intersections [29]. Besides that, these applications can benefit from historic traffic data [30]. Therefore, the traffic data has to be transferred to a central solution which is able to process and store this data in order to optimize the traffic flow for a whole city. This could be accomplished with an integration of cloud computing as we describe in the next section.

2.3 Integration of IoT with Cloud Computing

There are various differences between the two concepts of IoT and cloud computing. Table 2.1 shows the characteristics of both concepts. As can be seen, the characteristics are even complementary [4]. By an integration of cloud computing and IoT, both concepts can benefit from each other, which also fosters the development of new applications.

In general, the integration of the IoT and cloud computing offers multiple advantages. For example, as IoT devices are typically resource-constrained – particular in terms of computation and energy capacity [19] – the IoT could benefit from the cloud by using the virtually unlimited computational capabilities of the cloud. The processing of data analysis, collaborative applications, and likewise operations can be done in the cloud. With task offloading from the IoT to the cloud, the energy on IoT-devices can be saved

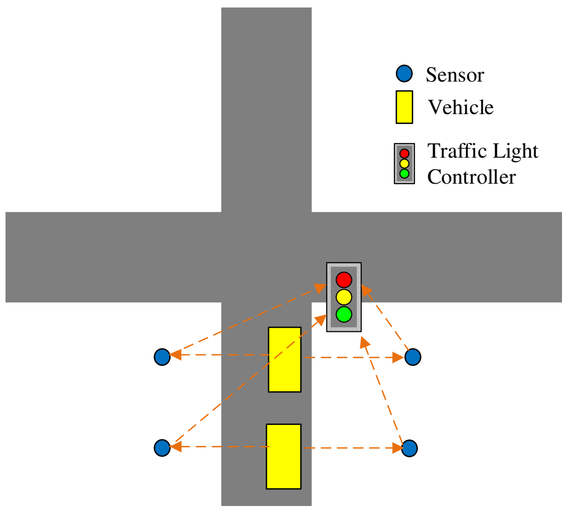


Figure 2.2: Traffic lights controlling application with sensors [27]

Table 2.1: Complementary aspects of cloud and IoT (from [4])

	IoT	**Cloud**
Displacement	pervasive	centralized
Reachability	limited	ubiquitous
Components	real world things	virtual resource
Computational capabilities	limited	virtually unlimited
Storage	limited or none	virtually unlimited
Role of the Internet	point of convergence	means for delivering service
Big data	source	means to manage

[4]. Furthermore, IoT devices are able to produce a huge amount of data. A combination of IoT and cloud computing offers new possibilities for data aggregation, integration, and sharing with third parties [4]. Besides that, the data generated by heterogenous IoT devices is often unstructured or semi-structured [31]. After transferring the data from the IoT devices to the cloud, the generated data can be stored and then retrieved in a homogeneous way from any place. Furthermore, security mechanisms can be applied centrally in the cloud. Securing the data on distributed heterogenous IoT devices is much harder [4].

Related to the previously mentioned example, Figure 2.3 shows the system architecture of a traffic light control application which combines IoT and cloud technologies [29]. Similar to Figure 2.2, vehicles are illustrated by yellow rectangles and the orange dashed lines represent the communication of the participating components. In contrast, the sensors are replaced with video cameras which record the traffic at the intersection and the cloud is part of the system architecture. Based on the videos and the detected vehicles, the schedule for the traffic lights is controlled to optimize the traffic flow. As vehicle detection is a complex task, this task is offloaded to the cloud. Besides that, the cloud stores and processes not only local data and events but also data from other intersections in a city. Therefore, with the integration of the cloud, the traffic lights control application is able to offer a broader solution.

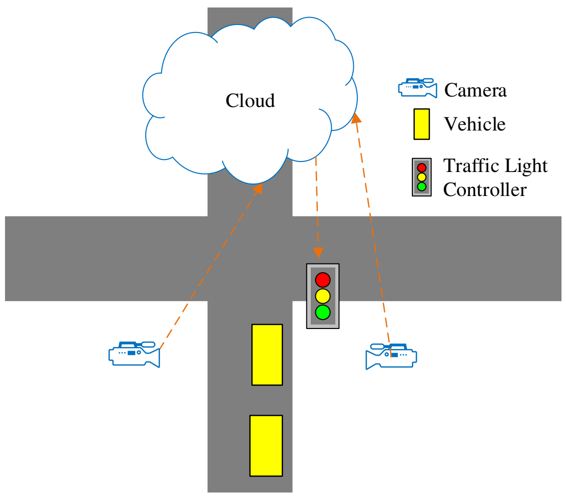


Figure 2.3: Traffic lights controlling application with cloud computing [29]

Nevertheless, integration of IoT devices and cloud computing has its challenges. As processing-intensive tasks are often offloaded to the cloud, the overall time for receiving the result is not only the processing time of the application at the cloud but also the time for the data transfer to and from the cloud [32]. Besides that, cloud-based applications can be distributed over different clouds. The communication between the devices and the cloud and the inter-cloud communication is done via the Internet. This all sums up in high traffic and results in high latency. However, a large amount of cloud-based applications are latency-sensitive. The latency could be that high that the delay is unacceptable and this leads to unusable applications [32].

To meet these challenges and provide low-latency, bandwidth-efficient, and resilient services to IoT applications, a new approach was established, referred to as edge computing. Edge computing aims at offering an extension for the cloud-based infrastructure but not to replace it. The goal is to increase the computing resources at the network edge and therefore place them closer to the user and the IoT devices [33]. A similar idea pursues fog computing. We discuss fog computing and its differences to edge computing in the following.

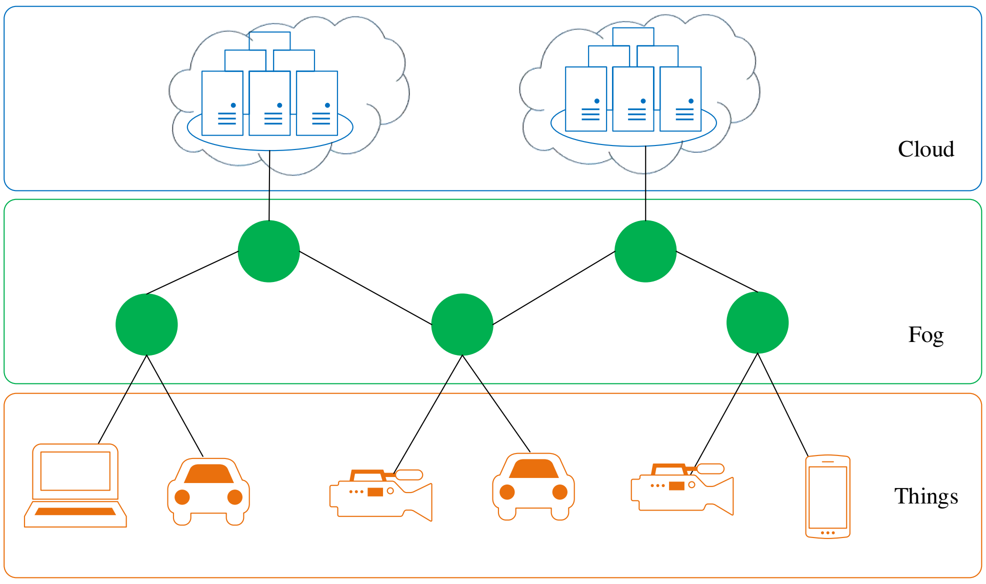


Figure 2.4: IoT and fog computing (based on [10])

2.4 Fog Computing

Fog computing provides compute, storage, and networking services between IoT devices and the cloud. It allows processing near the IoT devices by so-called fog nodes and enables processing at specific locations. Simply said, a fog is a cloud closer to the ground which is also the reason for the term. The processing is done closer to the ground, hence closer to the IoT devices and users [34]. The central data centers of the cloud are usually not in the vicinity of the user.

Similar to edge computing, fog computing also provides computing resources located at the edge of the network, but not exclusively [10]. The OpenFog Consortium – an organization which aims at creating an open reference architecture for fog computing and is now part of the Industrial Internet Consortium [35] – states that there are some key differences between edge computing and fog computing. Fog computing is an extension of cloud computing where the implementations of the architecture can be placed on multiple layers of a network's topology. According to the definition of the OpenFog Consortium, edge computing excludes the cloud and can be implemented only in a small number of layers. Besides computing resources – which are also provided by edge computing – fog computing further considers other things like networking or storage [36].

Figure 2.4 depicts an example of a fog structure. At the bottom are the smart things of the IoT. They are connected with fog nodes which are depicted as green filled circles. The fog nodes in turn can be connected to other accessible fog nodes or possibly different cloud computing providers.

Some main characteristics of fog computing are the following [10]:

Low latency One of the fundamental characteristic of fog computing is that IoT devices and end users benefit from placing the fog nodes near to or at the network edge. Hence, the fog nodes are in the proximity of the IoT devices and end users and are able to provide services with low communication latency.

Geographical distribution Fog computing aims at providing services and applications for a very large number of widely distributed IoT devices. As the fog nodes are located in the proximity to the IoT devices, the fog nodes are also distributed over a wide area.

Mobility The connected IoT devices are not always stationary. The communication with moving and mobile devices is essential for particular applications. Therefore, fog computing supports mobile techniques.

Heterogeneity The fog nodes are very different to each other related to their hardware and software. Besides that, the fog nodes are deployed in a variety of environments.

Related to the example of an intelligent traffic lights control application, Figure 2.5 shows an solution using fog computing [25]. Again, vehicles are illustrated by yellow rectangles and the orange dashed lines represent the communication of the participating components. In contrast to Figure 2.3, the traffic light controller is a fog device and communicates with the vehicles. Based on the gathered information, the traffic light controller itself decides – based on the received information – if the schedule for controlling the traffic light has to be adapted. Therefore, this solution offers low latency as the calculation and decision is made in the proximity of the traffic light and not in the cloud. Additionally, the data from the traffic light controller is sent to the cloud for global long-term storage and analytics [10].

Fog computing with its characteristics has the potential to tackle the challenges of IoT, cloud computing, and the integration of both concepts. Therefore, the research activity in the area of fog computing is high and several papers related to fog computing were published over the last years [37].

We presented a brief background information of IoT, cloud computing, and fog computing. Both, cloud computing and fog computing are essential parts of our proposed solution. Therefore, an understanding of these concepts is essential. Another important concept for reaching the goals of our approach is *erasure coding*. We discuss erasure coding in the next section in detail.

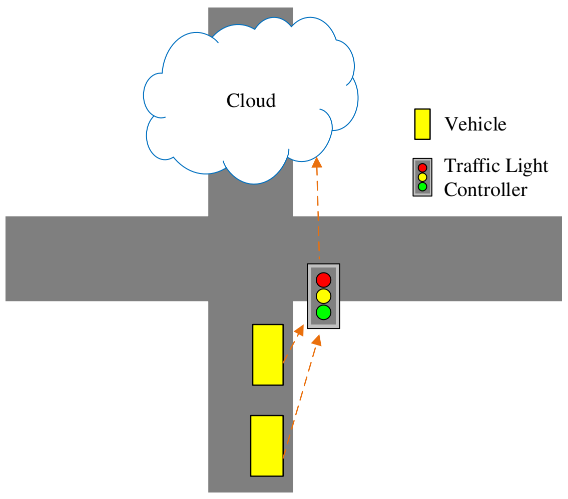


Figure 2.5: Traffic lights controlling application with fog computing [25]

2.5 Erasure Coding

As we have mentioned, the cloud storage providers offer highly available access to stored data which means that they have to provide mechanisms to cope with system failures – like data recovery. As data must not get lost at a failure of one component, the data needs to be stored somehow redundantly. In this section, we investigate possibilities to offer data redundancy.

A naïve approach for data redundancy would be to replicate the data to another component, i.e., copying the whole data [38]. Erasure coding offers a more economic approach.

Erasure coding splits a data block into m data chunks and encodes them into n chunks which are then stored on different disks. The original data block can be regenerated from any m chunks ($m < n$) [39]. Therefore, a system using erasure coding can tolerate $n - m$ storage failures. The parameters m and n define the used erasure coding configuration (m, n). For instance, a (2,3) erasure coding configuration means that one data block is split into 3 chunks in equal size, but for regeneration only 2 of them are needed. Figure 2.6 depicts an example of a (2,3) configuration with 3 blocks. Every block (A, B and C) is erasure coded into $m = 2$ data chunks and $n - m = 1$ parity chunk. We label the data chunks with d_1 or d_2 and the parity chunks with a p. The $n = 3$ chunks are stored at three different disks, namely s_1, s_2 and s_3. If one disk fails, all 3 blocks still can be regenerated as $m = 2$ chunks are available out of all 3 blocks.

With erasure coding, it is possible to establish RAID (Redundant Array of Inexpensive Disks) features [41] at cloud storage level. Furthermore, erasure coding is a superset of RAID and full replication systems, as these systems can be described by a particular erasure coding configuration [42]. For example, RAID-5 can be achieved by a (4,5) erasure coding configuration. An erasure coding configuration with $m = 1$ would be a system with full replication, e.g., a (1,3) configuration defines a full replication system

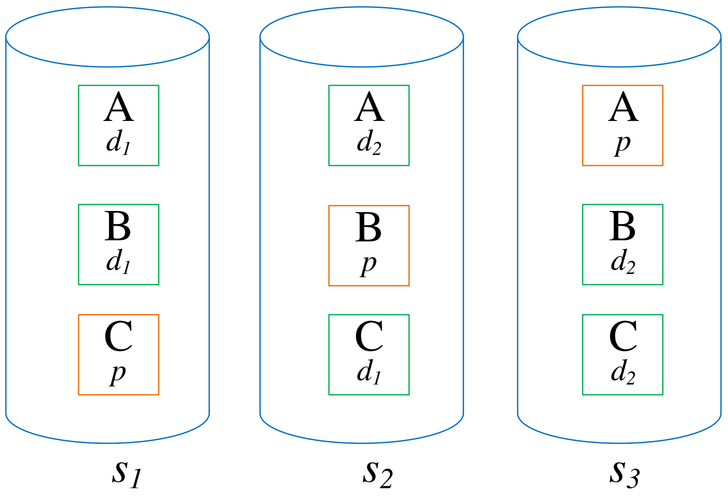


Figure 2.6: Example of a (2,3) erasure coding configuration (based on [40])

that replicates the data to three disks. As there is only the subset of m chunks needed to regenerate the original block and the summarized size of m chunks is approximately equal to the size of the original block, erasure coding with $m > 1$ can offer the same level of redundancy as full replication but with less storage space needed. More precisely, the storage overhead factor can be expressed as $\frac{n-m}{m}$, which describes the relation of parity chunks to data chunks [43]. Both, the erasure coding configuration for a full replication with two disks (1,2) and the erasure coding configuration (2,3) can tolerate one storage failure, but the storage overhead factor using the full replication ($\frac{2-1}{1} = 1$) is higher than by using the (2,3) erasure coding configuration ($\frac{3-2}{2} = 0.5$). Therefore, with the erasure coding configuration (2,3), less storage space is needed than with full replication.

There are various algorithms for erasure coding. Reed-Solomon codes have the longest history among these algorithms [43]. Multiple open-source libraries already exists that implement Reed-Solomon codes. Because of the computation complexity of Reed-Solomon codes, they were historically viewed as expensive. Based on today's techology, most Reed-Solomon implementations are fast enough that the bottleneck is located in another component than the processor, e.g., the disk I/O. [40]. Hence, we will use a Reed-Solomon implementation in our approach.

In this chapter, we presented fundamental background information related to cloud computing, the IoT, and fog computing. Besides that, we discussed the technique of erasure coding which is essential for our approach. In the next chapter, we discuss approaches of relevant related work and compare our approach to these approaches.

CHAPTER 3

Related Work

In this chapter, we present and investigate related work in the area of cloud-based and fog-based storage. As we have mentioned in Chapter 2, the research activity in the area of cloud computing and fog computing is very extensive. Therefore, several different approaches and techniques have been published in this area over the past years. In the following, we describe and discuss approaches which are closely related to our solution or use techniques and methods our solution is based on. For each approach, we mention the motivation of the approach, analyze the methods the authors use, and conclude with the potential limitations. First, we look at approaches in the area of cloud storage. After that, we discuss approaches that involve the storage and computational resources of the fog. We conclude this chapter with a comparison of the presented related work and our proposed solution.

3.1 Cloud Storage

There are various approaches which have been designed for implementing storage in the cloud. Many of these approaches aim at replicating data to different cloud storage providers with the goal of reducing the risk of being dependent on one particular cloud storage provider. This can be useful for the case that the resources of one cloud provider are temporarily or constantly unavailable, since the user could still access the data from another provider. Such approaches, which are closely related to our solution or use techniques and methods our solution is based on, are discussed below.

3.1.1 MetaStorage

MetaStorage uses a distributed hash table to replicate data to different cloud storage providers [38]. Therefore, the approach offers high availability and vendor independence.

A core component of the architecture is a middleware containing the distributor which provides the functionality to retrieve and replicate the data among available storage providers. Based on a prioritized list, the system replicates the data to different cloud storage providers. The middleware is designed to work in a highly scalable environment, as multiple instances of the distributor can be executed at the same time.

MetaStorage uses full replication which means that the whole data is replicated multiple times. This results in high amounts of utilized storage which leads to high costs as the needed storage capacity has to be paid at each used cloud provider.

3.1.2 SPANStore

SPANStore is a key-value store which replicates data to geographically distributed data centers [44]. Similar to MetaStorage, the main focus lies on providing availability of data. Besides that, SPANStore aims at lowering latency.

By using data centers of multiple different cloud storage providers, SPANStore is able to provide a geographically denser set of data centers than a single storage provider could do. Therefore, the system can move copies of data closer to the clients which results in lower latency. Since the approach replicates the data to different cloud storage providers, SPANStore is able to provide high availability and vendor independence. The way the system writes the copies of the data and where the system gets the data from is defined by the replication policy. A central component – the Placement Manager – computes the optimal replication policy. To find the optimal policy for a particular application, the authors present an optimization problem which does not only take latency into account but also the overall costs.

Although the costs – based on the pricing models of the cloud storage providers – are part of the optimization problem, the system uses full replication – similar to MetaStorage. Therefore, SPANStore has also the drawback of using high amounts of storage capacity which again leads to high costs.

3.1.3 CORA

Similar to SPANStore, CORA is a middleware which aims at reducing latency by finding an optimized data placement [14]. But differently, the approach takes also historic access latency information into account in order to calculate an optimized placement.

Additionally and in contrast to the previously presented work of SPANStore and MetaStorage, CORA uses erasure coding in the replication process. The data is split into multiple chunks which are then distributed to different cloud storage providers. Therefore, CORA is able to provide the same level of availability and vendor independence as SPANStore and MetaStorage but with smaller additional storage needed. CORA allows to define the required level of availability and vendor independence for each file separately. Besides latency, availability, and vendor independence, CORA optimizes the data placement to minimize costs based on the pricing models of the cloud storage providers.

As every cloud storage provider receives one chunk of data at most, no cloud storage provider has enough information to regenerate the original file. Since CORA stores all information about the data objects and chunks locally, the user has to trust the system. While the user can upload already encrypted data to CORA to preserve privacy, encryption is not an integrated part of the system.

3.1.4 TrustyDrive

TrustyDrive is a storage system which aims at preserving privacy and protecting the anonymity of the data in the cloud [2]. In this approach, neither the storage system itself nor a cloud storage provider is able to regenerate the original data.

As the system has not the required information to encode the data into multiple chunks, this has to be done at the client side by the user. After providing meta-data for the particular chunks, the user forwards the chunks to the dispatcher which computes further meta-data and stores the chunks at the cloud storage providers. The storage system ensures that one cloud storage provider gets at most one chunk. As the user is responsible for encoding the data, the user is free to choose a preferred encoding algorithm and can utilize encryption techniques to increase the anonymity. This makes the user layer of the system highly configurable. Based on the particular purpose, users can decide if encryption is required or the fact that no cloud storage provider has enough information to regenerate the original data is sufficient to secure the data. Furthermore, the level of availability and vendor independence depends on the chosen encoding algorithm. Consequently, the user layer is the only component of the system which is able to regenerate the original file.

Besides the advantages of passing the control over the level of availability and privacy to the user, this approach can lead to a leakage of sensitive data or, even worse, data loss if the user is not able to encode and encrypt the data correctly. Furthermore, TrustyDrive does not consider latency.

3.1.5 SSME

The Secure Storage in Multi-Cloud Environment (SSME) is an approach which focuses on providing a secure and privacy-preserving data storage in a worldwide distributed cloud environment [45]. The architecture of SSME consists of two main parts: The SSME client and the SSME middleware. Similar to TrustyDrive, the client is responsible for the encryption of the data and the SSME middleware guarantees that the encoded chunks of the data are distributed to different cloud storage providers. But in contrast to TrustyDrive, the authors specify the client component in detail and provide solutions for the secure communication between the client and the middleware. A key element of the approach is the use of a combination of both symmetric and asymmetric encryption.

The SSME middleware consists of two components, namely the SSME server and the Trusted Cloud Service. The SSME server is responsible for splitting and merging the data as well as for distributing and receiving the data chunks – to and from the cloud storage providers. The server itself does not store data locally – all processing works only in volatile memory. The second component of the SSME middleware is the Trusted Cloud Service. The Trusted Cloud Service offers authentication and a temporary data storage which is needed by the system during internal operations. Users of the SSME system are able to integrate cloud services they trust for the functionalities of the Trusted Cloud Service. This increases the confidence of the users in using the SSME system. Moreover, the practical usage of the approach is shown by implementing and integrating an Android App into the SSME System and therefore providing a secure storage on mobile devices in a multi-cloud environment [46].

Although the uploaded files are possibly encrypted, the complete files are stored at the Trusted Cloud Service for a particular time. The user has to find public cloud services to entrust with the storage of the data. Alternatively, the user can set up a private cloud for the functionalities of authorization and temporary data storage, but this leads to a higher effort.

3.2 Fog Storage

As we have mentioned in Chapter 2, cloud computing leads to multiple challenges and some of these challenges are inherited by cloud storage. As these challenges can potentially be tackled by using the storage and computing resources of the fog, many approaches and techniques have been presented in the area of fog computing over the last years. For instance, related to storage in the fog, the authors in [47] present FogStore, which is a key value store guaranteeing low latency and strong consistency for geo-distributed applications. A further storage service for the fog which is based on IPFS[1] is presented in [48]. Aiming at reducing latency, the authors in [49] present a popularity-based cache placement for the fog. More generally related to the fog infrastructue, the placement of services in the fog is both the content of [50] and [51]. Out of these large amount of approaches in the area of fog computing, we focus on approaches which are closely related to our solution or use techniques and methods which can help us to reach the goals of preserving privacy, reducing latency and offering high availability. In the following, we discuss these closely related approaches in detail.

3.2.1 iFogStor

iFogStor aims at reducing the overall storage latency in a fog infrastructure [52]. To this end, the authors present a data placement strategy which optimizes the overall storage and service latencies of devices and the latency between the devices in a fog computing infrastructure. To find an optimal data placement strategy is challenging, as a fog computing infrastructure typically means a geographically distributed and hierarchical placement of many heterogeneous devices.

[1]https://ipfs.io/

The authors define three types of actors in a fog infrastructure: a data host, a data producer, and a data consumer. The data host represents a node with storage capability which could be located in the fog or in the cloud. A data producer is able to output data. This producer could be a sensor which gathers data or a fog node which forwards the result after processing incoming data. The data consumer requests data for different purposes, such as storing the data or processing the data for further usage. The actors in a fog infrastructure can be classified as one or more of these types. Based on this classification and the determination of the latencies between the different devices and nodes in the network topology, a Generalized Assignment Problem (GAP) is defined. The authors present an exact and a heuristic solution for this as NP-hard known problem. Besides that, an outcome of this work is an overall software framework which allows to formulate the problem and implements the proposed solutions. Using the exact solution, the authors show that the overall latency is reduced by more than 86% compared to a cloud-only approach and by 60% compared to a naïve fog computing approach.

iFogStor aims at lowering latency but does not consider privacy or availability. Besides that, the different parameters, such as the role of the actors in the fog infrastructure and the latency between the actors have to be known in advance. If these parameters change, the optimization has to be solved again. This solving time can be unacceptable in case of large-scale application – at least with the exact solution. Furthermore, the evaluation is done in a software framework without real sensors or real fog devices. In a real environment the parameters would be more dynamically which would have an impact on the results.

3.2.2 Layered Distributed Storage

Konwar et al. present a two-layer architecture for a distributed storage [53]. The fog layer consists of nodes in the geographically proximity of the clients and allows to process data closer to the client. The powerful nodes of the cloud act as backend layer.

Typically for a storage system, the client is able to write data to the system and read data from it. The client interacts with the fog layer only. A write operation is finished after the data is written to the fog layer and does not wait for storing the data in the cloud. This speeds up the operation and leads to low latency. The fog layer uses erasure coding for splitting the data into multiple chunks. The fog nodes act then as virtual clients for the cloud layer and distributes the chunks to the cloud nodes. As the servers of the fog layer are restricted in their total storage capacity, the fog layer is used only as temporary storage. Data which is still in the temporary storage could be served directly from the fog layer. Hence, the fog nodes act as a cache for data. This helps to reduce the communication costs between the two layers. Besides lowering latency and reducing costs, the approach offers high availability. The presented algorithm is designed to tolerate crash failures of half of the fog nodes and one third of the cloud nodes.

The authors discuss and analyze the properties of the presented algorithm and show the advantages related to low latency and reduced costs of the approach formally. An implementation of the algorithm is not provided. Therefore, it is not possible to show an evaluation of this approach in a real world environment. Besides that, the approach does not consider privacy, which is a main aim of our approach.

3.2.3 Network Coded Distributed Storage

The work of Cabrera et al. aims at providing a solution for reliable storage and reliable communication in a fog of unreliable devices [54]. For traditional cloud storage data centers, it can be assumed that a failing node can be replaced by a new node in a short time frame. That is not suitable for a fog context, as nodes can be disconnected without a replacement. Besides that, the devices of the fog are much more likely to get damaged or run out of battery than the high-end devices of cloud storage data centers. This leads to a highly dynamic system. Therefore, the authors provide a protocol which allows that the nodes continuously leave the network one by one, whilst the remaining nodes maintain reliable access to the data.

The system is described as follows: An uploader chooses a subset of interconnected fog nodes and distributes the data to them. The members of this subset inform constantly the other ones that they are still connected. If a member disconnects, the remaining members start a repair session, i.e. start to interchange data. After this repair session, the system is able to survive another fail of a member node. A key point of this approach is the usage of network coding. Compared to the Reed-Solomon code, the same or less amount of data has to be transferred during the repair session. Another advantage is, that no node needs to decode the data packets to encode it again during the repair process. This results in less complexity.

The authors show that it is possible to maintain reliable access to data in a network of unreliable devices. The complete neglect of involving traditional cloud storage could lead to better latency but also to a data loss if all fog nodes fail. Besides that, the work does not consider privacy.

3.2.4 Three Layer Privacy Preserving Storage

Wang et al. present an approach to store the data in a three-layer architecture, where the cloud, the fog, and the user's local machine represent one layer respectively [55]. A main aim of this approach is to preserve privacy which is reached by using erasure coding and the storage and computational resources of the fog.

The approach uses erasure coding to divide the files into three parts. These parts differ in their size. The smallest part is stored at the user-side, the next bigger one in the fog and the biggest part of the data in the cloud. Each part alone has not enough information to regenerate the original data. The encoding information is stored only at the user's local machine and in the fog. Both parts of this system are controlled by the user. Based on the Reed-Solomon code, the authors introduce a Hash-Solomon code algorithm. This algorithm has an additional property which helps to improve the privacy protection: Before encoding the data to the different chunks, a hash transformation is done, which disrupts the original sequence of the data. Speaking of text documents, the default Reed-Solomon code would lead to chunks with partial information of the original document in the original sequence. If an attacker gets access to one or more chunks, the attacker could also get fragmentary information of the original file. This could be critical if the documents contain sensitive data. The Hash-Solomon code prevents this by randomizing the original sequence. The relevant hash information is part of the encoding information which is stored at the user's local machine and the fog node respectively.

The authors show that the approach is able to encode and decode data efficiently related to the processing time of the algorithm as well as to the needed cloud storage space. Besides that, the approach is proved to ensure privacy by providing a theoretical safety analysis. Nevertheless, the work does not aim at providing low latency or consider availability.

3.2.5 Secure Data Storage and Searching

In [56], the authors present a framework for data processing in the area of industrial IoT by integrating fog computing and cloud computing. A main aim of this approach is the usage of privacy-preserving techniques without decreasing the usability of the data, i.e maintaining the searchability of the data.

As the focus of this approach is the industrial IoT, the idea of using the system differs from the previously presented approaches. The industrial IoT devices produce data which is then transmitted to the fog nodes. The fog nodes act as proxies, as they process the data and forward it to the cloud. Users of the system connect directly to the cloud and get the required information in a preprocessed presentation without involving the fog nodes. Besides aggregation and storing, the fog nodes are responsible for the encryption of the data while maintaining the searchability.

Besides the main aim of preserving privacy, the approach offers low latency for the IoT devices by using the resources of the fog. The data is also highly available as it is stored in the fog and in the cloud. But since the data is already preprocessed by the fog node, the fog node and the cloud node have a different representation of the data respectively. This varies from our understanding of availability.

We analyzed and discussed the most relevant related approaches to our approach in detail. For this, we selected different approaches in the area of cloud computing and fog computing and described the motivation as well as the limitations of each approach. In the next section, we compare the presented approaches to each other and to our approach.

3.3 Comparison

In the last two sections, we presented approaches in the area of cloud computing and fog computing which are closely related to our approach or use techniques and methods our approach is based on. In this section, we compare these approaches to each other and to our approach. For this, we select various features of cloud-based or fog-based storage systems and analyze which of the presented approaches fulfill these features. In the following, we describe the selected features and discuss the importance of these features for a storage system which aims at preserving privacy, providing low latency, and offering high availability.

Availability This feature describes the capability of an approach to serve the requested data despite of a failure of one or multiple servers in the cloud or devices in the fog [14].

Latency Approaches which cover this feature use mechanisms to reduce the elapsed time between the request and the corresponding response at uploading or downloading data compared to a system which uses only one cloud storage service [53].

Privacy This feature is fulfilled by approaches which implement procedures to hide details of the data or details of the ownership from the storage providers and the public [55].

Coding Coding refers to techniques which are used to store data with the same level of availability as full replication but with reduced amount of needed storage capacity or with reduced data transfer [54].

Encryption Approaches which cover this feature use encryption mechanisms, so that the stored data can only be read with the correct secret key [45].

Cloud Storage This feature means that the corresponding approach uses the storage and computational resources of the cloud [14].

Fog Storage Approaches marked with this feature use the storage and computational resources of the fog [52].

Table 3.1 provides an overview showing the different presented approaches and their respective features. We differ between fullly-covered features and partially-covered features. Many of the presented approaches aim at providing high availability, often applying erasure coding. Only the approach in [54] uses network coding instead of erasure coding. Besides that, many approaches have the goal of low latency. Especially the approaches to fog-based storage can offer better latency results. Only some of the approaches offer the ability to preserve privacy. Furthermore, only a few approaches offer a high level of privacy using encryption.

As can be seen, our work is the only one which offers all listed features. We provide a solution which offers high availability using erasure coding and the reliability of cloud storage services, low latency by using the storage and computational resources of the fog, and high-level privacy by a privacy-preserving distribution of data in combination with encryption.

Table 3.1: Comparison of related work

✓	Feature is fully covered
~	Feature is partially covered
-	Feature is not covered

Feature	[38]	[44]	[14]	[2]	[45]	[52]	[53]	[54]	[55]	[56]	Our work
Availability	✓	✓	✓	~	✓	-	✓	✓	-	-	✓
Latency	-	✓	✓	-	-	✓	✓	✓	-	✓	✓
Privacy	-	-	~	✓	✓	-	-	-	✓	✓	✓
Coding	-	-	✓	~	✓	-	✓	✓	✓	-	✓
Encryption	-	-	-	~	✓	-	-	-	-	✓	✓
Cloud Storage	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	✓
Fog Storage	-	-	-	-	-	✓	✓	✓	✓	✓	✓

In this chapter, we presented the most relevant related approaches to our approach as well as a comparison of these approaches. The comparison shows the limitations of the presented approaches and that our approach can tackle these limitations. We present the detailed design of our approach – including the used methods and techniques – in the next chapter.

CHAPTER 4

Design

In this chapter, we present the design of our approach. We start with the analysis of the necessary requirements for a system that addresses the goals of preserving privacy, providing low latency, and offering high availability. Then, we present an architectural overview of the design and the behavior of the system and the participating components of our system. According to the formulated requirements, we discuss the Placement Strategy as well as our design decisions concerning privacy and security. Finally, we describe the communication between the participating components in detail.

4.1 Requirements

In this section, we summarize the requirements of our system. We split the requirements into *functional requirements* and *non-functional requirements*. The functional requirements describe the processes of uploading and downloading files which are essential to the system. The goals of our approach of preserving privacy, providing low latency, and offering high availability are part of the non-functional requirements.

4.1.1 Functional Requirements

We describe the functional requirements by providing use case descriptions. A use case is a sequence of actions performed by a system which leads to a result of value for the user [57]. Use cases are nowadays widely used and an effective way of identifying, implementing, and validating the functional requirements of a system [58].

In our proposed solution, we identify the processes of uploading and downloading as use cases. Both, at uploading and downloading files, the user gets a result of value – either the file is stored or the file is retrieved. In the following, we present the use cases based on a use case description template [59]. According to the template, the use cases are characterized by an actor which represents anything interacting with the system, e.g. a user or another system, a precondition which is an assertion about the state of the world at the beginning of the use case, and a postcondition which describes the state of the world after a successful completion of the use case. According to the template, the basic flow describes the steps which lead to a successful completion of the use case. Furthermore, the error flow describes the steps which do not lead to a successful completion, i.e. do not fulfill the postcondition.

Use Case: Upload

- **Actor:** User
- **Precondition:** The user has access to the user interface of the system and has successfully logged in with its username and password at the system.
- **Postcondition:** The user has received a file with all information to download the previously uploaded file which we call the *regeneration-file*.
- **Basic Flow:** The steps in the basic, hence successful, flow are:
  1. The user selects a file using the user interface.
  2. The user submits the selected file to upload it to the system using the user interface.
  3. The user receives a message about the successful upload and the regeneration-file.
- **Error Flow:** If an error occurs, the steps are:
  The steps 1. and 2. are the same as in the basic flow.
  3. The user receives a message with a reason for the failed upload.

Use Case: Download

- **Actor:** User
- **Precondition:** The user has access to the user interface of the system, has successfully logged in with its username and password at the system, and has the regeneration-file containing the needed information to get the original file from the system.
- **Postcondition:** The user has received the original file.
- **Basic Flow:** The steps in the basic, hence successful, flow are:
  1. The user selects the regeneration-file to get the related original file using the user interface.
  2. The user submits this regeneration-file using the user interface.
  3. The user receives a message about the successful download and the original file.
- **Error Flow:** If an error occurs, the steps are:
  The steps 1. and 2. are the same as in the basic flow.
  3. The user receives a message with a reason for the failed download.

4.1.2 Non-Functional Requirements

We present the non-functional requirements according to ISO/IEC 25010 which is a standard for modelling quality properties of a product [60]. This product quality model categorizes product quality properties based on various characteristics. To define the non-functional requirements, we identify which of these characteristics are applicable to our system, in order to achieve our goals of preserving privacy, providing low latency, and offering high availability.

Thus, in the following we identify these characteristics and for each one we define the relevant requirements. Therefore, the ISO/IEC 25010 standard helps us to create a structured view on the non-functional requirements and helps us to establish a suitable architecture for our system.

Performance Efficiency

As stated in the ISO/IEC 25010 standard, the performance efficiency describes the performance relative to the amount of resources used under stated conditions. In our system, this can be translated to two requirements:

- **Time Behavior:** The system should be able to provide low latency at uploading and downloading files. More precisely, the Round Trip Time (RTT) between sending a request and getting a response should be lower than using a cloud-only solution.
- **Resource Utilization:** Despite being highly available, the system and the data should use available storage space efficiently. Concretely, the system should need less space for storing data than a system with full replication, but offering the same level of availability.

Compatibility

According to the standard, compatibility describes the degree to which a system can exchange information with other systems in order to perform its required functions. For this characteristic we identify one concrete requirement for our system:

- **Interoperability:** The system should be able to exchange information with other components in a standardized way. More precisely, the components of the system should offer interfaces which implement a well-known standard. Furthermore, the interface definitions should be published and accessible to the participating components.

Security

The ISO/IEC 25010 standard describes security as the degree to which a system protects the data so that other persons or other systems have access to the data, according to their levels of authorization. In our system, security is related to the following three requirements:

- **Confidentiality:** The system should ensure that a user has access to a particular file only if this user has the corresponding regeneration-file. No other user or component should have access to the original content or a part of the original content of the file, as well as should not be able to reconstruct the original file, independent of the amount of effort.
- **Integrity:** The system should make sure that no component can modify the stored data which would lead to a change of the content of the original file.
- **Authenticity:** There is a defined set of users and components which are involved during the processes of uploading and downloading. The system should ensure that only these users and components are allowed to use the system.

Portability

As stated in the standard, portability describes the degree to which a system can effectively and efficiently be adapted for different environments. In our system, this can be translated into two requirements:

- **Installability:** The components of the system should be able to be installed and uninstalled on the most popular operating systems (e.g. Linux-based OS, Windows, etc).
- **Replaceability:** The fog components of the system should be able to replace each other, without the need of exchanging their own current states among each other. More precisely, if a component fails, another component should be able to take on the role of the failing component without knowing about the processed requests or the stored data of the failing component. Therefore, the fog component should be able to process the requests independently from each other without having information of the previously processed requests. Besides that, the stored data of the failing fog component should not be necessary for further processing.

Reliability

According to the ISO/IEC 25010 standard, reliability describes the degree to which a system component performs specified functions under specified conditions for a specified period of time. We identify two requirements related to reliability:

- **Availability:** The data should be highly available. More precisely, the availability of the data should not depend on one particular storage component, hence temporary or constant unavailability of one storage component should not prevent access to the uploaded files. Despite of a failure of at least one storage component in the cloud or the fog, the data should also be available and usable. Besides that, the data should also be available if the used fog node for uploading differs from the fog node which is used to download the data. This could be the case if the local network changes between upload and download, as particular fog nodes might be accessible only in the local network.

- **Recoverability:** The system should be able to recover from a failure of a component easily. More concretely, the system should be able to add new fog nodes, if one fog node fails, without needing information from the failing fog node. Therefore, the fog node should be able to handle requests independently from each other. Besides that, the stored data of the fog node has not be necessary for further processing.

After defining the requirements for our system, we are now going to present an architectural overview of our proposed system.

4.2 Architectural Overview

In this section, we present an architectural overview of a system which fulfills the previously formulated requirements. We refer to this system as *FogStorage*. Furthermore, we discuss the related components and the functionality of the FogStorage system. Figure 4.1 depicts the FogStorage system in an example setup. As can be seen, the FogStorage system uses components in the cloud layer, in the fog layer, and in the user layer. The cloud layer consists of services offered by cloud providers, the fog layer contains devices at the edge of the network, and the user layer consists of the user devices, e.g. a notebook or a smartphone.

The design and the implementation of the components in the user layer and in the fog layer are part of our approach. In Figure 4.1, we illustrate these components with light-green filled boxes. The component of the user layer is an application running on the user device which allows the user to connect to the FogStorage system as well as to upload and download files. We refer to this application as *FogStorage Client*. The fog layer is a middleware which receives and processes the requests of the FogStorage Client. We refer to this middleware as *FogStorage Service*. The FogStorage Service can be executed on one or multiple fog nodes. We refer to one fog node running a FogStorage Service instance as *fog storage node*.

In the cloud layer, our approach uses services from *cloud storage providers (CSP)* to store data chunks. Examples of cloud storage providers are Google with its Google Cloud Storage service[1] or Amazon with the Amazon S3 service[2]. Both mentioned providers offer multiple worldwide distributed data centers for their storage services. These data centers are isolated from each other and can be accessed independently. We refer to one

[1]https://cloud.google.com/storage
[2]https://aws.amazon.com/s3/

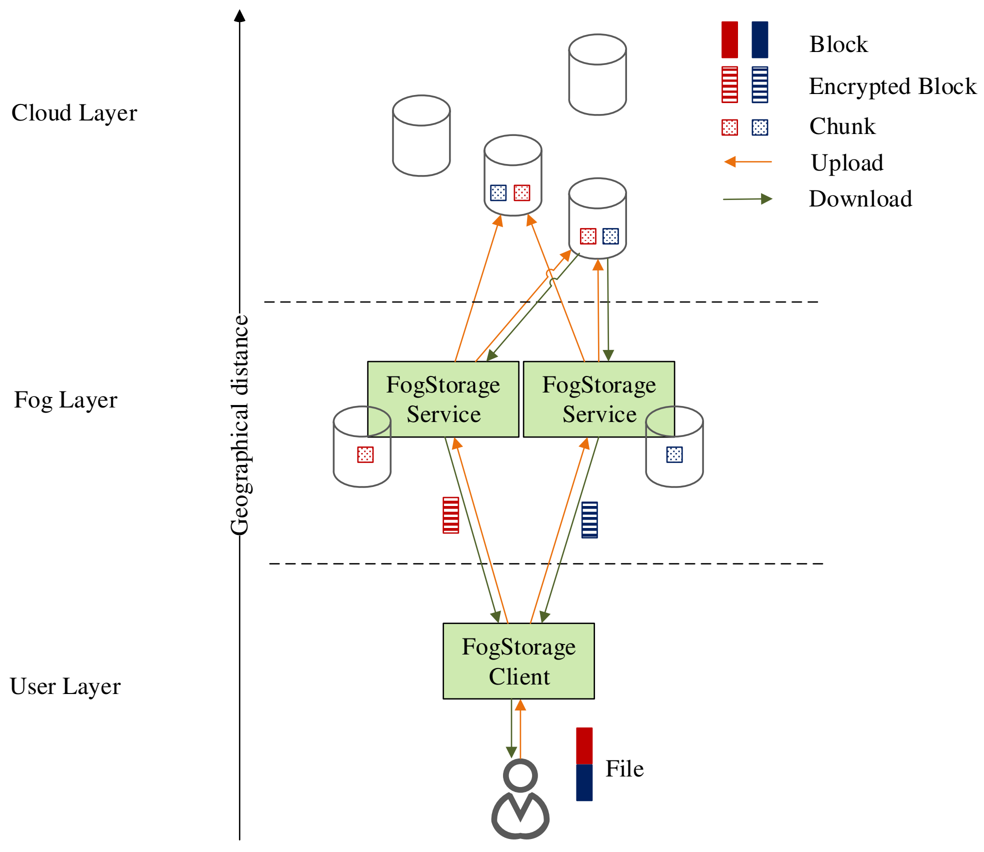


Figure 4.1: Architectural overview of the FogStorage system

data center as a *cloud storage node*. In Figure 4.1, we depict four accessible cloud storage nodes in the cloud layer.

Besides the different participating components, Figure 4.1 shows the communication between the components during the main use cases of uploading and downloading of files. We refer to the state of the participating components as *Upload Mode* and *Download Mode*, respectively. We illustrate the communication between the participating components in Upload Mode with orange arrows and in Download Mode with green arrows.

Concretely, in Upload Mode the user uploads a file using the FogStorage Client. This Client connects to one or multiple fog storage nodes, i.e. fog nodes which are running the FogStorage Service. After that, the Client splits the file into multiple blocks (blue and red boxes) which is done to simplify the handling of large files. We provide a detailed discussion of this process in Section 4.2.3. After splitting the file into multiple blocks, the Client encrypts the blocks (blue and red boxes with dashed filling) and forwards them to the FogStorage Service on the fog nodes. The Service applies erasure coding [39] for

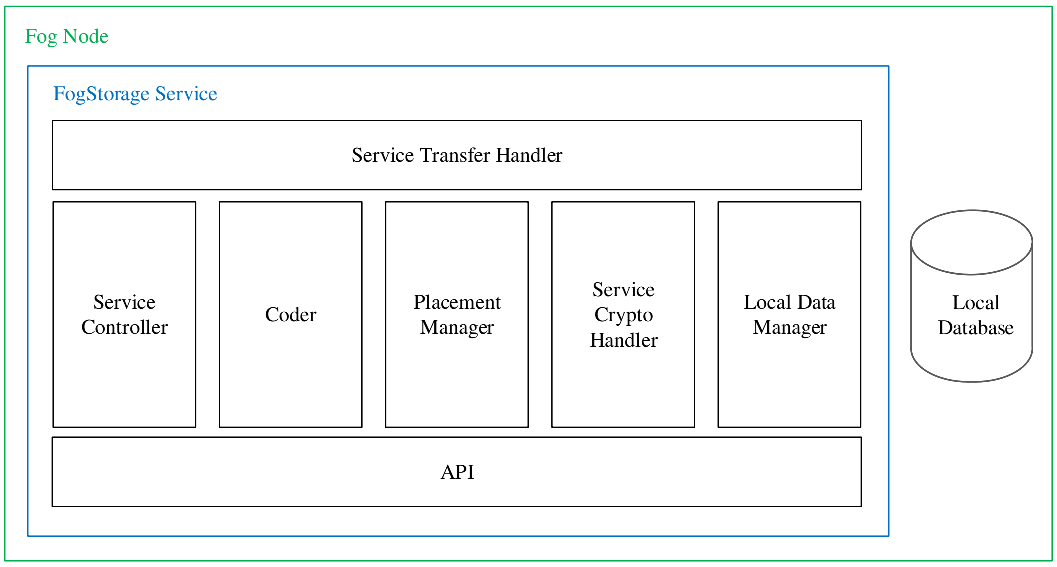


Figure 4.2: Architectural overview of FogStorage Service

redundant storage and therefore splits the data into multiple chunks (blue and red boxes with dotted filling) according to the chosen (m, n) configuration. In this illustration, a (2,3) configuration is used. After that, the FogStorage Service distributes two chunks to cloud storage nodes and stores the remaining chunk at the local disk.

In Download Mode the user requests an original file using the FogStorage Client. The Client connects again to one or multiple fog nodes running the FogStorage Service and requests the needed blocks for recreating the original file. The Service collects the minimal amount of chunks needed from the cloud and fog storage nodes and recreates the original block. The FogStorage Service then returns the block to the FogStorage Client. The Client decrypts the blocks and merges the blocks to the final original file. We discuss the details of the uploading and downloading processes in the upcoming sections.

In the following, we present details of the inner structure of the FogStorage Service, the FogStorage Client, and the FogStorage system. To make our system more comprehensible, we start with the FogStorage Service because knowledge of the structure, mechanisms, and interfaces of the FogStorage Service can help to better understand the logic of the FogStorage Client and the FogStorage system.

4.2.1 FogStorage Service

Figure 4.2 shows the architecture of the FogStorage Service. The FogStorage Service is composed of the following components:

API The FogStorage Service provides an API to interact with it. The interface is used by the FogStorage Client as well as by other instances of the FogStorage Service on different fog nodes. The FogStorage Client uses the API to upload and download files, other instances of the FogStorage Service uses the interface to transfer chunks to and from the fog node. Besides that, the API provides the functionality to authenticate the FogStorage Client and other FogStorage Service instances. The available API endpoints are described in Section 5.1 in detail.

Service Controller The Service Controller is the main component of the FogStorage Service. It coordinates the communication between the other components and the processing of requests and data. The Service Controller prepares and validates the requests and their parameters, such as the desired (m, n) erasure configuration. Furthermore, the Service Controller manages the information to connect and interact with all other participating components. More precisely, the Service Controller knows all accessible fog storage nodes and how to connect and interact with them. Similarly, the Controller knows the accessible cloud storage nodes and manages the access information for the cloud storage nodes, i.e. the credentials for them. Furthermore, the Service Controller manages the calculated latency of the accessible fog storage nodes and cloud storage nodes. As the FogStorage system can only be used by registered users, the Controller knows which users are registered and how the users can be authenticated.

Coder The Coder is the component of the FogStorage Service responsible for splitting and merging the data by applying erasure coding. Based on the parameters m and n, the Coder executes the erasure coding on the given block. The blocks are created by the FogStorage Client. This process is described in Section 4.2.2. In Upload Mode the Coder splits the block into multiple (n) chunks which are then forwarded to the Service Controller. Besides that, each created chunk gets assigned an unique identifier. For this purpose, UUIDs [61] are used, as there is no central authority and the identifier has to be unique concerning the whole system. In Download Mode, the Coder receives the necessary (m) chunks and recreates the original block from it which is then again forwarded to the Service Controller.

Placement Manager The Placement Manager is a central component of the FogStorage Service, as it implements the essential Placement Strategy. The Placement Manager decides where to put the chunks of data in Upload Mode and where to get the necessary chunks in Download Mode. Therefore, the Placement Manager contributes an important part to reach the goals of preserving privacy, providing low latency, and offering high availability. Besides that, in Upload Mode the Placement Manager generates the *placement-info* which is then forwarded to the FogStorage Client. This placement-info contains necessary information concerning the uploaded chunks and information on which cloud storage nodes or fog storage nodes the chunks are stored. Furthermore, the placement-info contains the size and the checksum of the original block. The checksum is a value which is needed to verify the integrity of a block. Typically, the checksum of data is represented by a hash value created by using a cryptographic hash function [62]. As the placement-info is needed to regenerate the original uploaded data, it has to be provided by the FogStorage Client in Download Mode. Due to privacy preserving and security reasons, the Service does not store anything of this data. We present further details about the placement-info and the Placement Strategy in Section 4.3.1.

Service Crypto Handler Before the placement-info is transferred to the FogStorage Client, this info is encrypted with a – to the Client known – secret. The exchange of this secret is described in detail in Section 4.4. The encryption of the placement-info is the responsibility of the Service Crypto Handler. Besides that, the Crypto Handler handles the encryption and decryption during the authentication process. This process is also described in Section 4.4 in detail.

Local Data Manager The Local Data Manager handles the local data on the fog node. This component is able to store chunks at the local disk and to retrieve chunks from the local disk. Besides that, the Local Data Manager manages the information of the stored chunks, such as the unique identifier, the chunk size, and the information of the remaining space on the local disk. For this purposes, the Local Data Manager uses the Local Database which is described below.

Service Transfer Handler The main task of the Service Transfer Handler is to transfer chunks to the different cloud and fog storage nodes. Therefore, the Service Transfer Handler is able to connect and interact with these components. More precisely, the Transfer Handler knows the provided interfaces of the cloud and fog storage nodes and implements these interfaces.

Local Database The Local Database contains all management information which is then retrieved from the different components of the FogStorage Service. The database runs on fog nodes and is not an integrated part of the Service. Therefore, the database makes sure that stored data gets not lost after a restart of the Service instance.

4.2.2 FogStorage Client

Figure 4.3 depicts the internal structure and components of the FogStorage Client. The FogStorage Client consists of the user interface, the Upload Service, the Download Service, the Client Crypto Handler, and the Client Transfer Handler. The details of these components are described in the following.

User Interface The FogStorage Client is running on the user's local device. The user interface allows the user to interact with the Client which mainly means uploading and downloading files.

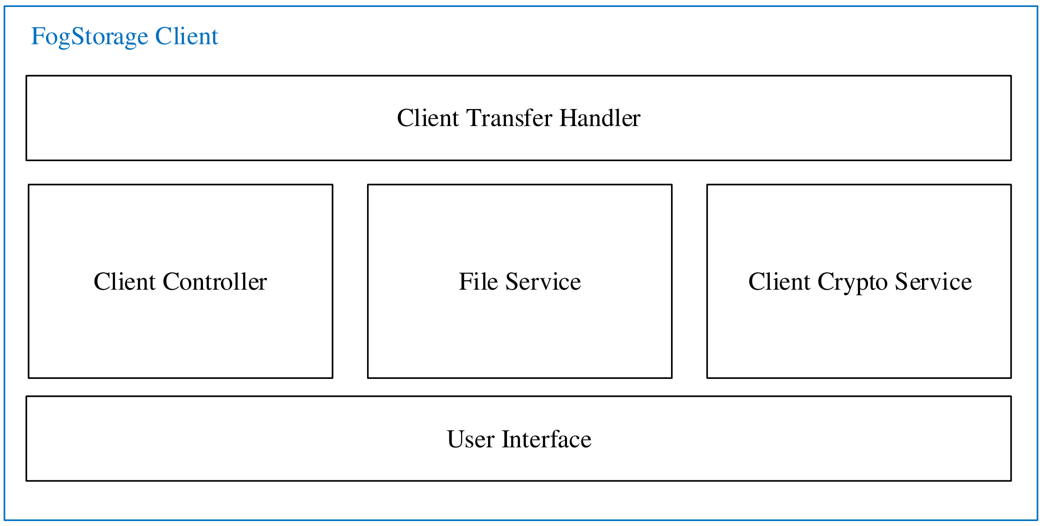


Figure 4.3: Architectural overview of FogStorage Client

Client Controller The Client Controller is the central component of the FogStorage Client. The Client Controller handles the requests of the user and prepares the request for further processing. Besides that, the Controller knows how to connect to the accessible instances of the FogStorage Service running on fog nodes.

File Service The File Service handles the upload and download of files. In Upload Mode, the service reads the file given by the user through the user interface and prepares the data for uploading to the FogStorage Service. This process contains the splitting of the file into multiple blocks. A block is a part of the original file with a particular size – the block size. We discuss details of the benefit of using blocks in Section 4.2.3. After creating the blocks, the File Service encrypts the blocks by using the Client Crypto Handler and uploads the blocks to one or multiple instances of the Service on accessible fog nodes. The Client Crypto Handler is also a component of the FogStorage Client and is described below. Besides that, the File Service creates the *regeneration-info*, which is needed to regenerate the original file in Download Mode. We discuss the regeneration-info in detail in Section 4.4.

In Download Mode, the File Service reads in the regeneration-info given by the user through the user interface, encrypts this information by using the Client Crypto Handler and forwards it to the FogStorage Service. After receiving the blocks from the FogStorage Service, the File Service decrypts the blocks using the Crypto Handler and merges the blocks to the original file which is then handed back to the user.

Client Crypto Handler The Client Crypto Handler is the central component for encryption and decryption tasks. These tasks consists of the already mentioned tasks during the upload and download process initiated by the File Service, as well as tasks during the authentication process. The authentication process is described in Section 4.4 in detail.

Client Transfer Handler The Client Transfer Handler of the FogStorage Client communicates with the interface of the FogStorage Service. It transfers the blocks of data to the Service in Upload Mode and requests and receives the blocks from the FogStorage Service in Download Mode.

4.2.3 FogStorage System

In this section, we discuss the architectural overview of the whole FogStorage system. As has already been mentioned in the introduction to this section, the FogStorage Service is executed on fog nodes which are typically unreliable devices and are more likely to get damaged or disconnected than the high-end devices of the cloud [54]. This leads to a highly dynamic environment.

We design and implement the FogStorage Service according to the Microservice Architecture, i.e. as a microservice, because this architectural style offers suitable approaches for that challenging environment. The Microservice Architecture has not a formal definition, but is defined by a set of characteristics [63]. Some key benefits of the Microservice Architecture which could help us in our environment, are resilience, scaling, ease of deployment, and optimizing for replaceability [64].

One characteristic of a Microservice Architecture is that the microservices are decoupled from each other and are able to produce a response to a particular request without being orchestrated by a central component. Therefore, the communication with the microservices is commonly done by using the simple REST protocol, instead of other complex protocols [63].

REST stands for *Representational State Transfer* and is based on four principles: Resource identification through a Uniform Resource Identifier (URI) [65], a uniform interface, self-descriptive messages, and stateless interactions [66]. Every request to a resource is stateless and completed after the response. Stateful interactions can only be done with explicit state transfers.

REST uses standardized technologies of the web for communication and messaging, like HTTP, XML, JSON or URIs. The uniform interface of REST consists of the well-defined and standardized HTTP methods. The most common are GET, HEAD, PUT, DELETE, and POST. Also the behaviour of these methods is standardized, e.g. GET retrieves data without changing it, DELETE requests the server to remove data, and PUT instructs the server to overwrite the data with other data [67]. Based on these standardizations, the communication with REST APIs and processing of messages, as well as the implementing of REST web services is simple, because most programming languages and operating systems support these technologies [66]. This also means that the various components of a system can be implemented in different technologies and programming languages but communicate via the standardized REST API. Many web services already provide REST APIs, e.g. the used cloud storage providers in our approach: Google Cloud Storage[3] and Amazon S3[4]. We present the different API endpoints of our system in detail in the implementation part in Section 5.1.

As has already been stated, in Upload Mode the FogStorage Client splits the submitted file into multiple blocks which is done because of following reasons: First of all, the fog nodes are possibly constrained regarding their storage space and are not able to process a big file at once. Besides that, if an up- or download of one block fails, the corresponding process has not to start from the beginning but can continue with the failing block at a later point in time. This would not be possible if the system would process the file at once, as the progress would be lost. But the most important reason for splitting the files into blocks is that it is possible to parallelize the processing of uploading and downloading of files. For example, in Upload Mode we can transfer two blocks to one fog node in parallel, as the further processing is more expensive than the splitting into blocks and the fog node has possible resources available. Thus, the fog node can process and upload the chunks of multiple blocks in parallel which would reduce the overall processing time. But as the fog node has surely not unlimited capacity, we can also use multiple fog nodes for processing. Due to the microservice architecture and the stateless characteristic of REST, every request is independent from each other. Therefore, it does not matter to which fog node and in which order the request for uploading or downloading is sent, the result is always valid.

We have given an architectural overview of the single components of our system as well as of the whole system. In the next sections, we dive deeper into the different parts and characteristics of our system. We start in the next section with the Placement which presents our Placement Strategy for distributing the chunks.

4.3 Placement

4.3.1 Placement Strategy

The core component of the FogStorage system is the Placement Strategy, which is implemented by the Placement Manager and therefore located at the FogStorage Service. Therefore, the upcoming definitions and algorithms are from the viewpoint of one particular FogStorage Service. The Placement Strategy is the most important part of our approach and is used for preserving privacy, providing low latency, and offering high availability. With respect to these goals and based on the chosen encoding parameters m and n of the (m, n) erasure coding configuration, the Placement Strategy finds a

[3]https://cloud.google.com/storage/docs/xml-api/overview
[4]https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html

suitable placement for n chunks in Upload Mode and retrieves the m best-placed chunks in Download Mode.

System Model

In this section, we define the system model and a formal definition of our problem finding the best Placement Strategy according to our goals of preserving privacy, providing low latency, and offering high availability. A detailed overview of the used notation is presented in Appendix A.

Chunks can either be placed at cloud storage nodes or at fog storage nodes, i.e. fog nodes which are running the FogStorage Service. As the Placement Strategy is implemented in the FogStorage Service and therefore the placement is done by a particular instance of the Service, only cloud and fog storage nodes can be considered which are accessible by this particular instance of the FogStorage Service. Of course, the fog node running this instance of the Service is also accessible and therefore a potential fog storage node.

We define a set of accessible cloud storage nodes C, where $c \in C = \{c_1, c_2, \ldots\}$ is a specific cloud storage node. Similarly, we define a set of accessible fog storage nodes F, where $f \in F = \{f_1, f_2, \ldots\}$ is a specific fog storage node. Furthermore, we define the set of all accessible storage nodes as S such that $S = C \cup F$, where $s \in S = \{s_1, s_2, \ldots\}$ is a specific storage node – either a cloud storage node or a fog storage node. Besides that, the FogStorage Service stores for every $s \in S$ the corresponding latency value l_s. Thus, the value l_{s_i} is the latency between the fog node executing the Placement Strategy and the accessible storage node s_i. The measurement of the latency values is part of the concrete implementation of the FogStorage Service. Therefore, we discuss the measurement and the management of latency values in Chapter 5.

As mentioned, every uploaded block is split into multiple chunks using erasure coding. Therefore, we define the set of chunks of a particular block as D, where $d \in D = \{d_1, d_2, \ldots\}$ is a specific chunk. The size of D depends on the given erasure coding configuration, i.e. $|D| = n$. For a successful regeneration of the original file, any subset of D with a size of m is needed.

Thus, the problem is to find one storage node $s \in S$ for every $d \in D$. Therefore, we define $\tilde{S} \subset S$ as the subset which contains the selected storage nodes for the placement, consequently $\tilde{S} = \{s_1, s_2, \ldots s_n\}$ and $|\tilde{S}| = |D| = n$. Furthermore, $\tilde{C} \subset C$ is the subset of the selected cloud storage nodes and $\tilde{F} \subset F$ the subset of the selected fog storage nodes, hence $\tilde{S} = \tilde{C} \cup \tilde{F}$. This means that every selected storage node has exactly one chunk. Multiple chunks on one storage node would conflict with the goal of tolerating failing nodes to achieve high availability. As the placement should fulfill the previously defined requirements, we discuss the needed constraints in the upcoming part.

Constraints

In this section, we define constraints for the Placement Strategy, such that the goals of preserving privacy, providing low latency, and offering high availability can be reached.

Privacy No single storage node should be able to regenerate the original file from the stored chunks and m chunks are needed to regenerate the original file. As we have defined in the last part by $|\tilde{S}| = |D| = n$, every selected storage has exactly one chunk. Therefore, the (m, n) configuration can only be valid if $m \geq 2$. This means also that full replication, where $m = 1$, is not possible. Besides that, the (m, n) configuration can only be valid, if $|S| \geq n$ holds.

Latency The goal and the requirement is that our approach offers low latency at uploading and downloading. Therefore, the Service has to distribute the chunks to the "nearest" storage nodes, i.e. the nodes with the least determined latency. Therefore, $\tilde{S}$ has to be the subset of S, where the sum of the corresponding latencies is minimal.

Availability As mentioned in Section 2.5, the (m, n) erasure coding configuration has to be chosen such that $m < n$. Therefore $m - n$ storage node failures, hence at least one storage node failure can be tolerated. This meets the requirement of having access to the data, despite a temporarily or constant unavailability of at least one storage. As mentioned, fog storage nodes are not as reliable as cloud storage nodes and are possibly only accessible in the local network. To fulfill the requirement of having access to data, independent from the uploading location, the Placement Strategy has to select at least m cloud storage nodes, hence $|\tilde{C}| \geq m$ and $|\tilde{F}| \leq n - m$.

Based on these constraints, the Placement Strategy has to find a subset $\tilde{S}$ of S with the size of n, where the sum of latencies is minimal and it holds that $|\tilde{C}| \geq m$ and $|\tilde{F}| \leq n - m$. Besides that, the Placement Strategy is only valid if the parameter $m \geq 2$.

Assumptions

Before we present the algorithm for our Placement Strategy, we discuss the assumptions we made. First of all, we assume that there are enough cloud storage nodes available according to the chosen (m, n) configuration. This means that there are at least m cloud storage nodes if also fog storage nodes are available and that there are at least n cloud storage nodes if no suitable fog storage node is available. This could be the case, if there is no fog storage node accessible for the Client, or the accessible fog storage nodes do not have enough free space for the chunks to distribute. We assume that cloud storage nodes do not have the restriction of insufficient free space, as cloud storage providers offer practically unlimited space.

Besides that, we assume that the information to accessible cloud storage nodes, i.e. the credentials for accessing the cloud storage nodes, are available to the FogStorage Service. Furthermore, we assume that the Service has the information to accessible fog storage nodes and the necessary information to connect to them. The distribution of this information could be done manually or by a trusted central authority. Nevertheless, the distribution of this information is not part of our approach. Similarly, we assume that the information to accessible fog storage nodes and Service instances are available to the FogStorage Client.

Placement Algorithm

Based on the requirements, the defined constraints, and the made assumptions, we present the algorithm for the Placement Strategy in the following.

After receiving a block in Upload Mode and erasure coding this file into multiple chunks, the Service has to find the best-suited storage nodes for the encoded chunks according to the previously defined Placement Strategy. In Algorithm 4.1, we show an algorithm to find the best-suited storage nodes, i.e. the placement for the chunks. The input parameters of this algorithm consists of the defined set of accessible cloud storage nodes C, the set of accessible fog storage nodes F, m and n of the (m, n) erasure coding configuration, as well as the chunk size b in bytes. As every chunk has the same size after erasure coding. It is sufficient to provide the size of any chunk to the algorithm.

First, the sets of C and F are combined to the set of all accessible storage nodes S (Line 1). After that, the algorithm checks if the (m, n) configuration is valid according to the previously defined constraints (Line 4). The method *orderByLatency* (Line 6) returns the storage nodes of the set S based on their latency in ascending order. Therefore, the set $S' = \{s'_1, s'_2, \ldots\}$ is an ordered set according to the latency of the storage nodes, i.e. $l_{s'_i} \leq l_{s'_{i+1}}$. As long as we do not have enough storage nodes – concretely n storage nodes – we check for every storage S' if it is in the set of C or it is in the set of F, hence the algorithm distinguishes between cloud storage nodes and fog storage nodes. While a cloud storage node is directly added to the selected cloud storage nodes $\tilde{C}$ (Line 13), the selection of a fog storage node is subject to the defined constraints. The check of the fog storage node according to the defined constraints is done by the algorithm in Line 9. More precisely, the fog storage node can only be selected if there are less than $n - m$ other fog storage nodes already in the set of selected fog storage nodes $\tilde{F}$ and the fog storage node has enough free space. Therefore, the algorithm calls the method *hasEnoughSpace* with two parameters. The first parameter is the current fog storage node s'_i, the second parameter b is the size of the chunk to place on the fog storage node. The method *hasEnoughSpace* returns *true*, if the fog storage node has enough space for one chunk of the given size b and *false* otherwise. Finally, the algorithm checks if enough storage nodes were found and returns an error if not (Line 19). Otherwise, the algorithm merges the both sets $\tilde{C}$ and $\tilde{F}$ to the resulting set $\tilde{S}$ containing the selected storage nodes. This set $\tilde{S}$ is part of the in Section 4.3.2 described placement-info which is returned to the FogStorage Client, thus to the user (Line 22).

Algorithm 4.1: Placement in Upload Mode

```
Input: A set $C = \{c_1, c_2, \ldots\}$ of accessible cloud storage nodes, a set
       $F = \{f_1, f_2, \ldots\}$ of accessible fog storage nodes, an integer $b$ representing
       the size of a chunk in bytes, an integer $m$ and an integer $n$ according to
       the chosen $(m, n)$ erasure coding configuration
Output: A set $\tilde{S}$ containing the selected storage nodes according to the
        Placement Strategy
1  $S \leftarrow C \cup F$ ; $\tilde{C} \leftarrow \emptyset$
2  $\tilde{F} \leftarrow \emptyset$
3  $x \leftarrow n$
4  if $(m < 2)$ or $m \geq n$ or $n > |S|$ then
5      return error: 'The $(m, n)$ configuration is not valid'
6  $S' \leftarrow orderByLatencyAsc(S)$
7  foreach $s'_i \in S'$ do
8      if $x \geq 1$ then
9          if $s' \in F$ and $|\tilde{F}| < (n - m)$ and $hasEnoughSpace(s'_i, b)$ then
10             $\tilde{F} \leftarrow \tilde{F} \cup s'_i$
11             $x \leftarrow x - 1$
12         else if $s \in C$ then
13             $\tilde{C} \leftarrow \tilde{C} \cup s'_i$
14             $x \leftarrow x - 1$
15     else
16         break
17     end
18 end
19 if $x \geq 1$ then
20     return error: 'No valid placement found'
21 $\tilde{S} \leftarrow \tilde{C} \cup \tilde{F}$
22 return $\tilde{S}$
```

In Download Mode, the Service receives the placement-info from the FogStorage Client. As this placement-info contains the chosen storage nodes, thus $\tilde{S}$, the Service has to find a subset S^* of $\tilde{S}$ which consists of the best-suited storage nodes for retrieving the needed chunks. Therefore, $S^* \subset \tilde{S}$ has to be the subset of $\tilde{S}$, where the sum of the latencies is minimal. The size of the subset S^* depends on the chosen (m, n) erasure coding configuration, as m chunks are needed to regenerate the original block. Therefore, it holds that $|S^*| = m$ and $|\tilde{S}| = n$.

The algorithm to find the subset S^* of $\tilde{S}$ is shown in Algorithm 4.2. The input parameter of this algorithm consists of the set of accessible storage nodes S, the set $\tilde{S}$ containing the during Upload Mode chosen storage nodes, as well as m and n of the (m, n) erasure coding configuration.

At first, the algorithm checks if there are enough accessible storage nodes at all (Line 3). Similar to the algorithm in Upload Mode, the set S is ordered according to their latencies using the method *orderByLatency* in Line 5. The algorithm collects storage nodes into the set S^* until the size of the set S^* is equal to m (Line 8). Finally, the resulting set S^* is returned (Line 14).

Algorithm 4.2: Placement in Download Mode

Input: A set $S = \{s_1, s_2, \ldots\}$ of accessible storage nodes, a set $\tilde{S} = \{\tilde{s}_1, \tilde{s}_2, \ldots \tilde{s}_n\}$ of storage nodes which were chosen during Upload Mode, an integer m and an integer n according to the chosen (m, n) erasure coding configuration

Output: A set S^* containing the storage nodes to retrieve the necessary chunks

```
1  $x \leftarrow m$
2  $S^* \leftarrow \emptyset$
3  if $m > |S|$ then
4      return error: 'Not enough storage nodes available'
5  $S' \leftarrow orderByLatencyAsc(S)$
6  foreach $s'_i \in S'$ do
7      if $x \geq 1$ then
8          $S^* \leftarrow S^* \cup s'_i$
9          $x \leftarrow x - 1$
10     else
11         break
12     end
13 end
14 return $S^*$
```

In both modes, namely Upload Mode and Download Mode, the algorithms return a set of best-suited storage nodes based on the given accessible storage nodes and the defined Placement Strategy. After the algorithms have finished, the FogStorage Service tries to connect to the storage nodes in the resulting set and executes the desired action, thus uploading or downloading the chunks. If an error occurs during these executions, the particular erroneous storage e is added to a set $E = \{e_1, e_2, \ldots\}$ of erroneous storage nodes. After that, the Placement Strategy has to find an alternative solution. In both modes, the algorithms are executed again with a reduced set $S = S \setminus E$. This happens as long as a valid placement is found or an error occurs. Errors can occur because the selected storage node is not accessible or the chunk can not be stored or retrieved from the selected storage node. The set E of erroneous storage nodes is only used for the particular request. Every request starts with an empty set $E = \{\}$. To avoid a large set of erroneous storages for every request, the concrete implementation of the FogStorage Service has to ensure that constantly inaccessible storage nodes are not part of the accessible storages S in the first place.

As mentioned, in Upload Mode the placement-info is the outcome of the algorithm, in Download-Mode a part of the placement-info is an input parameter. In the upcoming section, we present the structure and content of the placement-info in detail.

4.3.2 Placement-Info

In Upload Mode, the outcome of a successful placement is the placement-info. We have already discussed the content of the placement-info shortly in the description of the Placement Manager in Section 4.2.1. The placement-info contains all information which is necessary to regenerate the original uploaded block. Listing 4.1 shows the structure of the placement-info in JSON format by using example values. In detail, the placement-info consists of the following parts:

Upload Parameter This part contains the encoding parameters for the erasure coding configuration, which are set in the upload request (Line 2). The information consists of the number of data chunks (m) and the number of parity chunks ($n - m$).

Block Metadata Another part of the placement-info is the information of the uploaded block, but not the file name of the corresponding file or anything else which could give a hint of the content of the corresponding file. Concretely, the metadata consists of the block size in bytes (Line 5) and the checksum of the block (Line 4).

Chunk-Node Placement This part contains the information where to find the needed chunks to regenerate the original file. That is just a mapping of the used storage nodes with the encoded chunks. This mapping contains some additional metadata for the chunks and nodes which is described in the next two points.

Chunk Metadata Another part of the placement-info is the chunk metadata. This metadata contains the unique identifier of the chunk (e.g. Line 7), the chunk size in bytes (e.g. Line 8), and the index of the chunk (e.g. Line 9). The index i is an integer $0 \leq i \leq n$ and assigned by the Coder (cf. Section 4.2.1). As the order of chunks is essential to regenerate the original file, the Coder needs the index in Download Mode.

Node Metadata Actually, the node metadata contains only the identification of the used node for uploading the chunk (e.g. Line 10). The node could be a cloud storage node or a fog storage node. The identification is arbitrary but all instances of the FogStorage Service have to agree on these ids. For instance, this could be done by a central authority which assigns the ids to the available nodes and distributes them to the instances of the FogStorage Service. Anyway, this process is not part of our approach, as we assume that every instance of the Service has the same information regarding available nodes and their ids.

Listing 4.1: The *placement-info* structure

```
{
 "uploadParameter": { "dataChunks":2, "parityChunks":1 },
 "blockInfo":
   {"checksum":"44735018847d8b9cf25362b3c157464a",
    "size":2097184 },
 "placement":[
    {"chunk":{"uuid":"cc3cfc46-e58a-43b3-b169-c936da968469",
              "size":1048594,
              "index":0 },
     "node" :{"id":"AWS_Frankfurt" } },
    {"chunk":{"uuid":"9b08aeed-f9f1-4b53-a95b-21546b383664",
              "size":1048594,
              "index":2 },
     "node" :{"id":"GCP_Iowa" } },
    {"chunk":{"uuid":"a6e8de4e-23cc-4057-a2ed-070f087218c3",
              "size":1048594,
              "index":1},
     "node" :{"id":"FOG_A" } }
  ]
}
```

We have described how the placement of chunks is necessary to reach the goals of our approach, namely preserving privacy, providing low latency, and offering high availability. Nevertheless, a suitable placement is not sufficient to reach these goals, in particular privacy, to the desired degree. Therefore, we are going to discuss further privacy and security mechanisms of our approach in the next section.

4.4 Privacy & Security

Besides preserving privacy, which is a main goal, security in general is a significant part of our approach. The concrete security requirements related to confidentiality, integrity, and authenticity are described in Section 4.1. To fulfill these privacy and security requirements we utilize well-known methodology in the area of cryptography and combine symmetric and asymmetric encryption. We present the concerning mechanisms of our approach in the following.

4.4.1 File-Encryption

As shown in Section 4.3, our Placement Strategy fulfills the requirement of not having a single storage node being able to regenerate the original file. Nevertheless, every storage node stores a chunk of the original block. Regarding a block whose original file is a plain text file, a chunk could contain sensitive data which would be readable to everyone who has access to the storage. Therefore, the Placement Strategy is not enough to meet the requirement that no component, other than the FogStorage Client, should be able to get the content of a file, or a part of the content.

To this end, the FogStorage Client utilizes symmetric encryption before uploading to the FogStorage Service. More precisely, the FogStorage Client uses the Advanced Encryption Standard (AES) which is a standard for encryption specified by the NIST (U.S. National Institute of Standards and Technology) [68]. The AES is a symmetric block cipher algorithm which processes data blocks of 128 bits with possible key lengths of 128, 192, or 256 bits. Based on the chosen key length, the actual algorithm is reffered to as AES-128, AES-192, or AES-256. The algorithm was developed by Daemen and Rijmen and therefore originally called Rijndael algorithm [69].

The FogStorage Client uses the AES-256 algorithm and therefore generates a random key with a 256 bit length which is used to encrypt every block of the uploaded file. This key is then again needed to get the clear text in Download Mode, as a decryption of an AES-256 encrypted file without a key is practically not feasible [70]. Therefore, the FogStorage Client adds the key to the regeneration-info which is discussed in detail in the following section.

4.4.2 Regeneration-Info

As mentioned in the architectural overview in Section 4.3, the FogStorage Client creates the regeneration-info which is needed to regenerate the original file and is stored only by the user and not by any other component of the FogStorage system. In fact, this information is a plain text file containing the needed information. Listing 4.1 shows the structure of the regeneration-info in JSON format by using example values. In detail, the regeneration-info consists of the following parts:

Upload Parameter The block size defines the size of the block which can be chosen by the user in Upload Mode. As the block size has to be available in Download Mode to regenerate the file, the block size is part of the regeneration-info (Line 3).

File Metadata This part contains information of the uploaded file, such as the file name (Line 6) and the file size (Line 7) which is needed to regenerate the original file.

List of Chunk-Node Placements As we have discussed in Section 4.3.2, the FogStorage Client gets one placement-info for every upload request, hence for every uploaded block. In Upload Mode, the FogStorage Client collects these placement-info and adds them to the regeneration-file (Line 9). As in Download Mode, the FogStorage Client retrieves every block by using these placement-info and regenerates the original file, the order of the blocks is essential. In Upload Mode, the FogStorage Client has to ensure that the order of the placement-info is correct related to the uploaded blocks.

Secret Key Essential for the regeneration is, as described previously, the AES-256 key to decrypt the downloaded blocks. Therefore, the Base64-encoded key is part of the regeneration-info (Line 14).

Listing 4.2: The *regeneration-info* structure

```
 1 {
 2   "uploadParameter":{
 3      "blockSize":2097152
 4   },
 5   "fileMetaData":{
 6      "fileName":"TestData.zip",
 7      "size":10485760
 8   },
 9   "placementList":[
10      { ... },
11        ...
12      { ... }
13   ],
14   "secretKey":"B90D72432E07C43DE4840CC405B26935
15                B9D033C24281011DD7ED140C536C3636"
16 }
```

The regeneration-info has all the information which is needed to regenerate the original file. If the regeneration-info gets lost, the original file can not be retrieved from the FogStorage system. The FogStorage System itself does neither have enough information to recreate the regeneration-info nor to regenerate the original file. This is a desired characteristic of the system to reach the goal of preserving privacy. Therefore, the FogStorage system can not offer a mechanism to regain access to the corresponding original file of a lost regeneration-info. Furthermore, the original file can be retrieved from everyone who is in possession of the regeneration-info. Access from other users to the original file can be intentionally, hence to share the original file, but also unintentionally if the owner is incautious. Of course, as the regeneration-info is just a plain text file, it can be encrypted or distributed to other secure storage nodes. These methods and available possibilities to store the regeneration-info securely are not part of our approach, as the cautious usage of the regeneration-info by the user is assumed.

We have described the encryption of the uploaded files to increase the degree of privacy. In the next section, we discuss the mechanisms to prevent access for non-authorized users and components.

4.4.3 Authentication

Since the FogStorage system aims at supporting multiple users in parallel, the FogStorage system has to cope with multiple instances of the FogStorage Client on different devices which in turn communicate with multiple instances of the FogStorage Service on different fog nodes. These FogStorage Service instances communicate again with other Service instances and with cloud storage providers and their cloud storage services. Every component in the FogStorage system has to be sure that the interacting components are the ones which they claim to be. Otherwise, data and access to storage nodes might fall into the wrong hands. Therefore, authentication is an important part of our system to establish privacy and security.

We use a combination of symmetric encryption and asymmetric encryption for our authentication purposes. As symmetric encryption we use again AES-256, as asymmetric encryption our approach uses the RSA (Rivest, Shamir, Adleman) cryptography which is a public-key cryptosystem, hence one key of the generated key pair is publicly available [71]. More precisely, data which is encrypted by using the public key can only be decrypted by the owner of the key pair, as the owner is the only one who should have access to the private key – the second key of the key pair. Besides that, messages can be signed by the owner of the key pair using the private key and everyone else can verify this signature by using the corresponding public key. Similar to the AES algorithm, the strength of the RSA algorithm depends on the chosen key length. Currently, the NIST recommends a 2048 bit key for the RSA algorithm [72] which also our approach uses.

The Transport Layer Security (TLS) protocol which is used for communicating securely over the Internet, also uses RSA as method to exchange a secret key which is then used for symmetric encryption and decryption [73]. TLS is designed to prevent eavesdropping or tampering of messages and is widely used in the Internet. In fact, more than 50% of the top 1 million sites on the Internet offer communication via TLS [74]. So while the chances are high that the communication with the cloud storage providers is secure, we can not assume that every fog node supports the TLS protocol. This has to be considered in the authentication design process which we present in the following.

Authentication of a FogStorage Client against a FogStorage Service

We have already mentioned that every FogStorage Client has the necessary information to connect to the FogStorage Service instances which also contains the public key of the corresponding FogStorage Service. In our approach, we do not define how the FogStorage Client gets this information. This could be done by a manual distribution or by a trusted central authority. Nevertheless, we assume that the user and the FogStorage Client can trust this information. Similarly, the distribution of the information about valid users and their passwords is also not part of our approach. We assume that the Service has access to the user information and that the user information is stored securely. In Figure 4.4, we depict the login process of a user.

As every FogStorage Service instance is independent from other instances, the authentication process has to be done for every FogStorage Service instance separately which should be participating in Upload or Download Mode. We describe the steps for one particular FogStorage Service instance in the following.

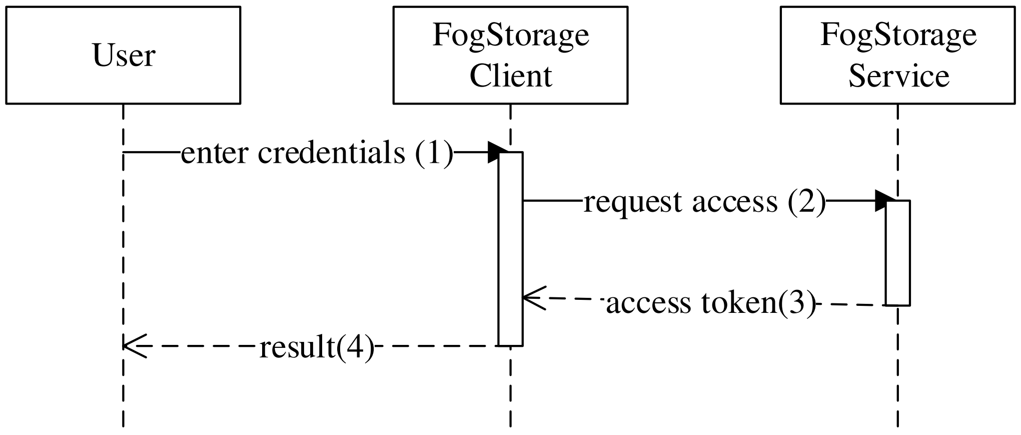


Figure 4.4: Authentication process for a user

1. First, the user is requested to enter the user name and the password. Every user has a user name which lets him identify uniquely at the FogStorage system and a corresponding password. After the user has entered the credentials, the user submits them to the FogStorage Client. The process to get these credentials, hence a registration process, is not part of our approach.

2. As the FogStorage Client gets the request to authorize the user at the FogStorage system, the Client generates a random AES-256 key. This key is intended to be shared with the FogStorage Service and therefore will be referred to as *shared secret*. The Client stores this shared secret for this particular Service instance. Besides the shared secret, the Client generates a random integer, which is later needed to verify the identity of the FogStorage Service. This random integer and the user password are then encrypted by using the shared secret. After that, the Client encrypts the shared secret with the public key of the FogStorage Service. Finally, the Client request access to the FogStorage system by transferring the following values to the FogStorage Service: The username in clear text, the encrypted password, the encrypted random integer, and the encrypted shared secret. Both, the password and the random integer are encrypted with the shared secret. The shared secret itself is encrypted with the public key of the FogStorage Service.

3. After the Service receives the access request, the Service decrypts the shared secret with its own private key. After a successful decryption, the shared secret can be used to decrypt the password and the random integer. The FogStorage Service can now check if the user is a valid one and the password is correct. If that is the case, the FogStorage Service can trust the user and therefore generates an access token.

This access token consists of the user name, the encrypted shared secret, and the validity of the token. Basically, the encrypted shared secret is just the submitted value of the FogStorage Client, i.e. the shared secret encrypted with the public key of the FogStorage Service. The validity specifies how long the access token is valid and when a new authorization process has to be initiated. Besides that, the Service signs the access token with a token key. This token key is neither the shared secret nor the private key, but is only known by the particular FogStorage Service instance. The resulting signature of the sign process with the token key is then added to the access token and becomes a part of the access token. Finally, the Service returns the access token along with the random integer in clear text to the FogStorage Client.

4. Receiving the access token, the FogStorage Client first checks if the received random integer is equal to the integer which was added at the access request in Step 2. If that is the case, the FogStorage Client can trust the particular FogStorage Service instance, as no other component could decrypt this integer. Besides that, the FogStorage Client stores the access token for the FogStorage Service instance until a new authentication has to be started, might because of the expired validity. At the end, the user is informed about the authentication process, either with a success or an error message.

At the end of the authentication process, the FogStorage Client has an access token as well as the shared secret. The access token is added to every request with the particular FogStorage Service instance. As the FogStorage Service has signed the access token with a token key only known by itself, the FogStorage Service can detect a modification of the access token and therefore reject the request in that case. The shared secret is needed by the FogStorage Client to encrypt information before transferring it to the FogStorage Service. As the public key-encrypted shared secret is part of the access token, the FogStorage Service gets the shared secret at every request and is able to decrypt the transferred information. Again, it has to be said that the access token as well as the shared secret are only valid for the particular FogStorage Service instance.

Authentication of a FogStorage Service against another FogStorage Service

Similar to the authentication of a user is the authentication process of one access-requesting FogStorage Service instance at another access-granting Service instance. We have already mentioned that every FogStorage Service instance has the necessary information to connect to other instances of the FogStorage Service which also contains the public key of the corresponding FogStorage Service instance.

Compared to the user authentication process (cf. Figure 4.4), Step 1 and Step 4 are not part of the authentication process of a Service instance, as a Service instance does not have a password and does not have to enter credentials. Much more, a Service instance initiates an authentication process by itself.

We shortly present the points, where the Service authentication process differs from the user authentication process:

No Password Instead of a password, the access-requesting FogStorage Service instance signs the random integer with its own private key and uses the resulting signature as alternative for the password. Besides that, the process is the same. The signature is encrypted with the shared secret and the shared secret is encrypted with the public key of the access-granting FogStorage Service instance. After decrypting the signature by using its own private key, the access-granting instance can verify the signature with the public key of the access-requesting instance. Therefore, the access-granting Service instance can trust the access-requesting Service instance.

No further use of the shared secret As described, the shared secret is used to en- and decrypt the sensitive information of the Service authentication process. After that, the shared secret is not used anymore, as the data exchanged between multiple instances of the FogStorage Service is either already encrypted or not sensitive at all. Therefore, the access token issued by the access-granting Service instance does not contain the shared secret.

After the authentication process, the access-requesting FogStorage Service instance has the access token, which can be used to communicate with the particular access-granting FogStorage Service instance. Similar to the user authentication process, the authentication process has to be done with every other FogStorage Service instance separately.

Authentication of a FogStorage Service against Cloud Storage Providers

Besides the authentication process with the FogStorage Service as access-granting entity, there is also an authentication process with external components, namely the cloud storage services of the particular providers. The credentials and communication information for the cloud storage services have to be available to all FogStorage Service instances and have to be the same for all FogStorage Service instances. The distribution of this information can be done manually or with a central instance. Nevertheless, this distribution is not part of our approach.

The specific authentication process with the cloud storage providers differs from one provider to another. Therefore, the concrete authentication is part of the implementation of the FogStorage system.

During all these authentication processes, sensitive data is needed by the FogStorage Service, e.g. private keys, the token keys or the credentials for the cloud storage providers. We do not present a generic method to store this data in a secure way. Hence, the cautious handling of the sensitive data is the responsibility of the concrete FogStorage Service implementation.

We have discussed the different functionalities of the participating components, as well as various mechanisms of our approach, including the Placement Strategy and the needed cryptographic mechanisms. We conclude this chapter with the communication protocol in the next section, where we discuss the processes in Download Mode and Upload Mode.

4.5 Communication Protocol

Finally, we present the communication protocols in the FogStorage system. The communication protocols consist of the communication between the FogStorage Client and the FogStorage Service, between the Service and the cloud storage services, as well as the communication between the multiple instances of the Service. This should bring together all parts of the design which we have already presented in detail in the previous sections.

4.5.1 Upload Mode

In Figure 4.5, we depict the communication between the different components of the FogStorage system in Upload Mode. The participating components are: One instance of the FogStorage Client, multiple (1 to n) FogStorage Service instances and multiple (1 to n) cloud storage nodes. The number on the arrows in Figure 4.5 correspond to the steps of the communication protocol which we describe in the following:

1. The upload is initiated by the user by submitting a file to the FogStorage Client for uploading it to the FogStorage system

2. The FogStorage Client splits the file into multiple blocks, encrypts them and forwards them to the FogStorage Service. This happens as long as there are blocks available. As Figure 4.5 shows, the FogStorage Client is able to contact multiple instances of the FogStorage Service for the different blocks, e.g. one block is uploaded to Service 1, another block is uploaded to Service n, etc.

3. The FogStorage Service receives the block, erasure codes it according to the upload parameters and calculates the placement of the resulting chunks. Based on the placement, the Service uploads chunks to other fog storage nodes (a), cloud storage nodes (b), and/or stores one chunk at the local disk.

4. The fog storage nodes (a) or cloud storage nodes (b) store the received chunk at their storage and returns a success message if the operation was successful or an error message otherwise.

5. If not all uploads were successful, the FogStorage Service adapts the placement according to the Placement Strategy. Otherwise, the Service returns the placement-info to the Client.

6. The FogStorage Client combines the placement-info of the different upload requests and additional data of the original file to the regeneration-info which is then returned to the user.

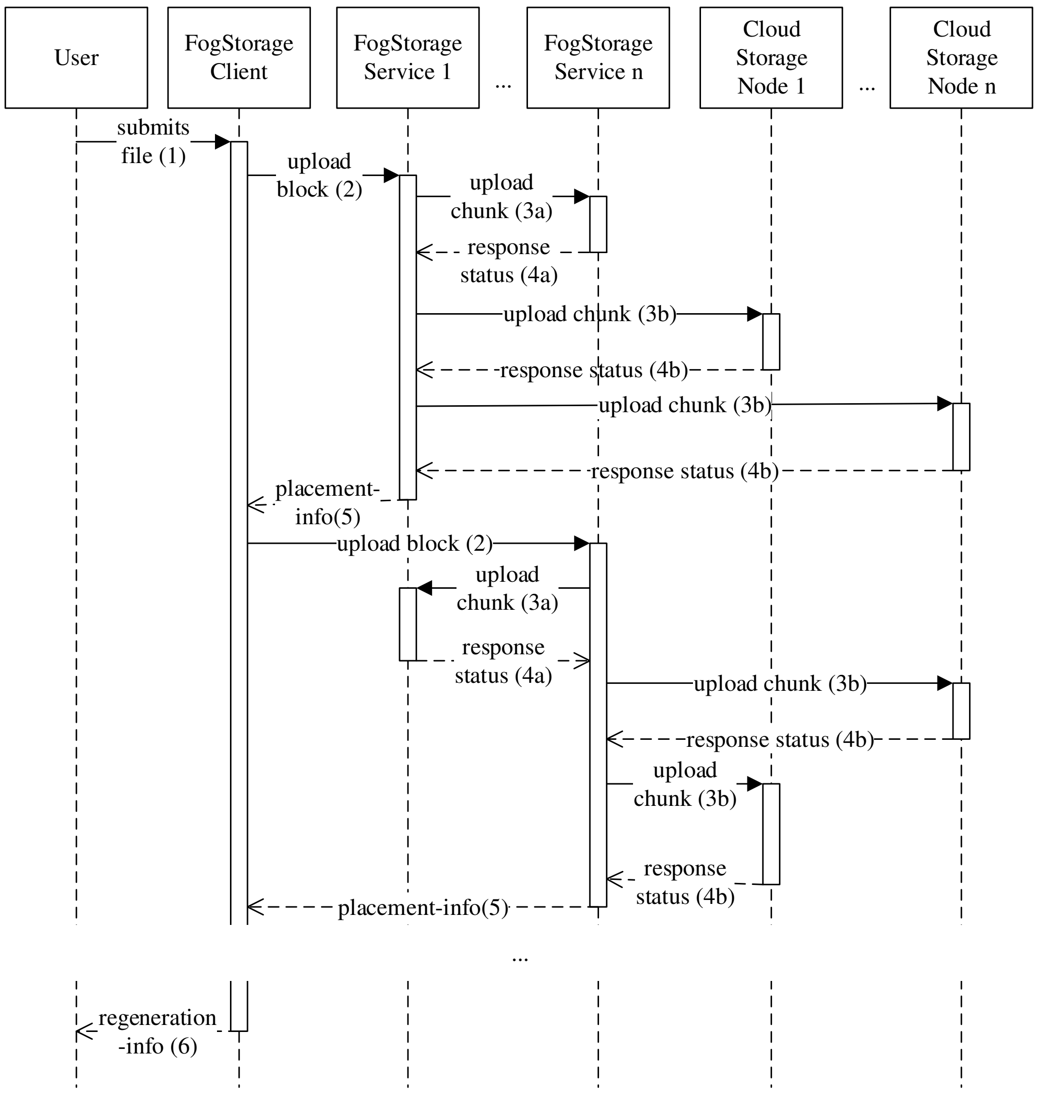


Figure 4.5: Communication of the different components in Upload Mode

Figure 4.5 shows the communication during the upload process in an exemplary manner, as a file usually consists of many blocks which have to be uploaded. Besides that, the figure suggests that the uploading of the blocks happens sequentially. But much more, Steps 3 and 4 are processed in parallel.

4.5.2 Download Mode

Similarly to the Upload Mode, we depict in Figure 4.6 the communication between the different components of the FogStorage system in Download Mode. The participating components are again: One instance of the FogStorage Client, multiple (1 to n) FogStorage Service instances, and multiple (1 to n) cloud storage nodes. The number on the arrows in Figure 4.6 correspond to the steps of the communication protocol which we describe in the following:

1. The upload is initiated by the user by submitting the regeneration-info to the FogStorage Client for downloading the corresponding original file from the FogStorage system.
2. The FogStorage Client extracts the multiple placement-info from the regeneration-info and requests the blocks from one or multiple FogStorage Service instances. If an accessible FogStorage Service instance is contained as fog storage node in a placement-info, the request will be forwarded to this particular instance. Similar to the Upload Mode, the Client is able to contact multiple instances of the FogStorage Service to retrieve the different blocks.
3. The FogStorage Service receives the request for downloading one block. Based on the placement-info and according to the Placement Strategy, the Service retrieves the needed chunks from fog storage nodes (a), from cloud storage nodes (b), or from the local disk.
4. The fog storage nodes (a) or cloud storage nodes (b) return the requested chunk from their storage if the chunk is available there or return an error message.
5. If not all downloads were successful, the FogStorage Service adapts the placement according to the Placement Strategy and tries to contact other storage nodes for downloading the corresponding chunks. Otherwise, the Service merges the received chunks using erasure coding to the original block and returns the block to the FogStorage Client.
6. The FogStorage Client merges the different blocks to the original file and returns the file to the user.

Again, Figure 4.6 shows the communication during the download process in an exemplary manner, as a file usually consists of many blocks which have to be downloaded.

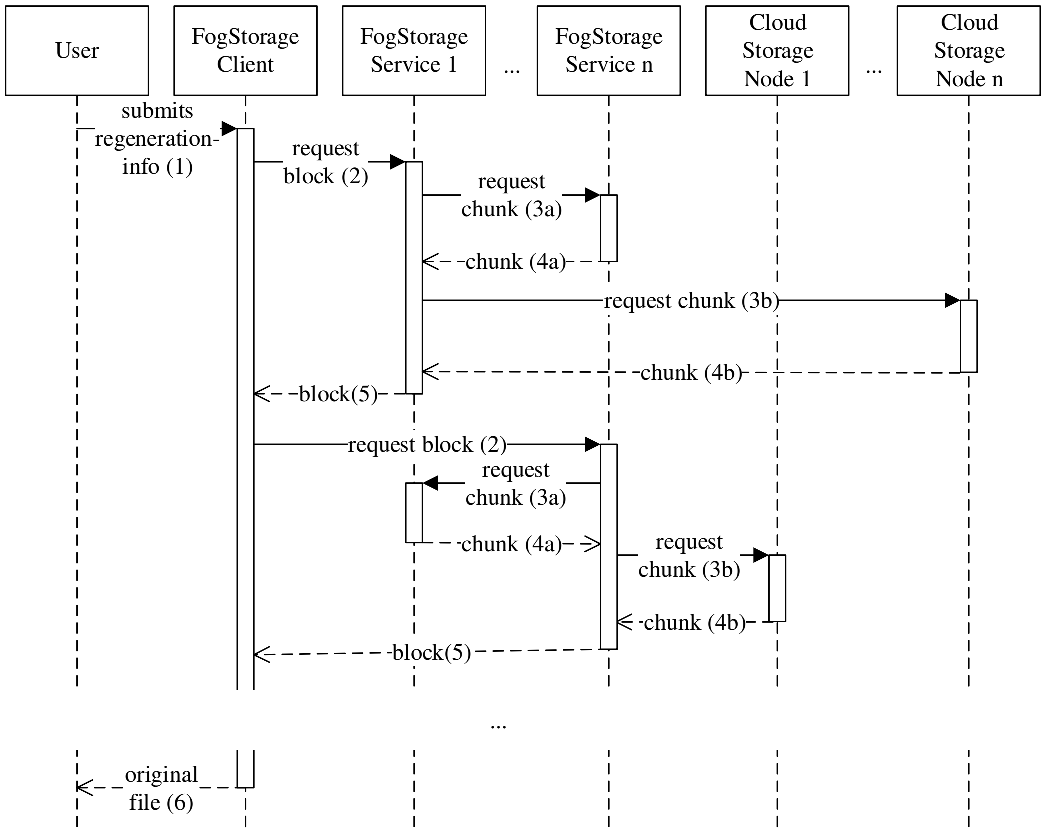


Figure 4.6: Communication of the different components in Download Mode

Furthermore, the figure suggests that the downloading of the blocks happens sequentially, but again, Steps 3 and 4 are processed in parallel.

This chapter describes the design of our approach. Based on the presented requirements, we have shown the needed functionalities and mechanisms to reach the goals of preserving privacy, providing low latency, and offering high availability. In the next chapter, we present a concrete implementation of the designed FogStorage system.

CHAPTER 5

Implementation

Based on the design of the FogStorage system which we have discussed in the last chapter, we present a concrete implementation in this chapter. Similarly to the structure of Chapter 4, we discuss the implementation of the FogStorage system in two separate sections which refer to the two components of the system, namely the FogStorage Service and the FogStorage Client. We start with the implementation details of the FogStorage Service and show details of the FogStorage Client implementation afterwards.

5.1 FogStorage Service

In this section, we present implementation details of the FogStorage Service. Based on the design described in Chapter 4, we focus on details which are not part of the design but important for the concrete implementation. We start with a presentation of the used technology stack consisting of the underlying technologies of the FogStorage Service and the used libraries and tools. After that, we present the implementation details for each component of the FogStorage Service.

5.1.1 Technology Stack & Libraries

The FogStorage Service is developed with Java 8[1] and the Spring Framework[2]. The Spring Framework offers a programming and configuration model which abstracts the infrastructural view and lets the developers focus on the business logic of the application. We use Spring Boot[3] as foundation for the development. Spring Boot helps to configure the Spring application for the particular requirements. With Spring Boot, it is possible to create stand-alone Spring applications with embedded web servers, hence the applications

[1]https://www.java.com/
[2]https://spring.io/projects/spring-framework
[3]https://spring.io/projects/spring-boot

do not need to be deployed to an external web server but can be executed on every Java running device. As dependency management and build tool we use Maven[4].

5.1.2 Implementation Details

According to the defined components of the FogStorage Service, we discuss the implementation details for each component in the following.

API As mentioned in Section 4.2.3, the FogStorage Service offers a REST API which we document with Swagger[5]. Swagger generates a documentation for the API endpoints based on the implementation. Hence, the documentation of the endpoints is always up-to-date. Besides that, Swagger offers a user interface which provides access to the documentation as well as a test functionality. With this functionality, it is possible to test the related endpoints directly by using a web browser. Based on the information of the Swagger documentation, we present the endpoints of our application. Since the details of the API are not essential for the understanding of the FogStorage Service, the details of every endpoint are presented separately in Section 5.3. In the following, we shortly discuss the main functionality of the different endpoints.

- **Authentication of users** The API of the FogStorage Service provides an endpoint to authenticate the users of the system. The parameters as well as the possible responses are presented in Table 5.1.
- **Authentication of other FogStorage Service instances** Besides the authentication of users, the authentication of other FogStorage Service instances is executed by calling an API endpoint. The details of this endpoint are shown in Table 5.2.
- **Upload of data blocks** An essential functionality of the FogStorage Service is the upload of blocks which is done by calling an API endpoint. The parameters and possible responses are described in Table 5.3.
- **Download of data blocks** Similarly to the upload of blocks, the functionality of downloading of blocks is done through by calling an API endpoint. The details of this endpoint are presented in Table 5.4.
- **Upload of chunks** FogStorage Service instances upload chunks to other FogStorage Service instances. This is done by calling an API endpoint. Table 5.5 shows the details of this endpoint.
- **Download of chunks** Similarly to the upload of chunks, FogStorage Service instances download chunks from other FogStorage Service instances. This is done through an API endpoint which is described in Table 5.6.

[4]https://maven.apache.org/
[5]https://swagger.io/

- **Requesting free space** As FogStorage Service instances need to know which other FogStorage Service instances qualify for uploading chunks to them, the free memory space has to be known. The free space of a FogStorage Service instance can be requested by calling an API endpoint. The details of this endpoint are described in Table 5.7.

Service Controller As discussed in Chapter 4, it depends on the concrete implementation how the information of accessible storage nodes as well as the information of registered users and their passwords is distributed to the particular FogStorage Service instances. In our implementation, we use a JSON file for this purpose. In Listing 5.1, we show an example of such a file. The file consists of two parts: The first part contains a list with available storage nodes – cloud storage nodes as well as all fog storage nodes. Every storage node has a unique name (e.g., Line 2), a type (e.g., Line 3), and a URL (e.g., Line 4). The current implementation supports three different types: AWS, GCP, and FOG. AWS refers to the cloud storage nodes of the Amazon Web Services (S3), GCP to the cloud storage nodes of the Google Cloud Platform (Cloud Storage), and FOG to fog storage nodes. Fog storage nodes are fog nodes executing the FogStorage Service. The URL contains the information how the related storage node is accessible. Additionally, the public keys (e.g., Line 11) of the fog storage nodes are part of the information. The second part contains the information of the registered users. This part contains all available users with their unique usernames (e.g., Line 17) and the password as a hash value (e.g., Line 18). The password is hashed by using the *bcrypt* hashing algorithm which prevents attacks with lookup tables [75].

Listing 5.1: Information to storage nodes and users

```
1  { "nodes": [
2     { "name": "AWS_Frankfurt",
3       "type": "AWS",
4       "url": "https://fogstorage-frankfurt.s3.eu-central-1.amazonaws.com/" },
5     { "name": "GCP_EU",
6       "type": "GCP",
7       "url": "https://storage.cloud.google.com/fogstorage-eu/" },
8     { "name": "FOG_A",
9       "type": "FOG",
10      "url": "http://184.172.8.1:8081",
11      "publicKey": "MIIBIj....hpczHNiwIDAQAB" },
12    { "name": "FOG_B",
13      "type": "FOG",
14      "url": "http://184.172.8.3:8081",
15      "publicKey": "MIIBIj....A/S+xK8iwIDAQAB"}  ],
16   "users": [
17    { "name": "user",
18      "pass": "$2a$06$...E011U1.k." }  ] }
```

The FogStorage Service is able to react to changes of the JSON file at runtime and therefore additional storage nodes and users can be added without restarting the FogStorage Service. To achieve this, the file has to be the same at every running FogStorage Service instance and therefore has to be synced when changed.

As can be seen in Listing 5.1, the latency of the various storage nodes is not part of the JSON file. The latency is not defined manually but determined dynamically during the execution of the FogStorage Service. The Service accesses a particular file on the different storage nodes and measures the time between sending the request and getting a response – the Round Trip Time (RTT). This time is stored in the Local Database. Similarly, the latency of the fog storage nodes is calculated, but additionally the fog storage nodes add also the remaining free space to the response. Both values, latency and free space, are then stored in the Local Database. If a storage node is not accessible or the request's response has an error message, this storage node is marked as invalid and not taken into account for further processing. The Controller takes a measurement of the latency in a periodical interval which can be configured for every FogStorage Service instance.

Coder For erasure coding, the FogStorage Service uses the Backblaze Java implementation of a Reed-Solomon erasure code[6]. This open-source library is as fast as a C implementation and can be integrated in a Java application, as it is also purely written in Java [76].

Placement Manager In our implementation of the FogStorage system, the Placement Manager with the implemented Placement Strategy is loosely coupled to the remaining system. That means that we can add further Placement Strategies by adding another implementation of it – without changing other parts of the system. Currently, the FogStorage system has two implemented Placement Strategies: The Placement Strategy with the corresponding algorithm from Section 4.3.1 as well as another Placement Strategy which behaves similarly but without including fog storage nodes, i.e., a cloud-only placement. The reason for this is that we can examine the benefits of using fog storage nodes compared to a cloud-only solution. Especially, the cloud-only approach is based on the storage system called CORA [14] – which we described in Chapter 3. We discuss the comparison in detail in Chapter 6. The Placement Strategy to choose for a request, is determined by the related request parameter.

As described in Chapter 4, the Placement Manager creates a checksum for every uploaded block. In our implementation, we use MD5 as the hash function to calculate the checksum [58].

Service Crypto Handler Both, symmetric and asymmetric encryption, are done with the default Java cryptography library. For the AES-256 encryption, we use the Cipher Block Chaining (CBC) operation mode which needs an Initial Vector (IV) [77]. As the

[6]https://github.com/Backblaze/JavaReedSolomon

Initial Vector does not have to be secret but needs to be unpredictable, we generate a random 16 byte long IV for every message to encrypt and add the IV to the encrypted message.

Local Data Manager The Local Data Manager abstracts the view on the Local Database as well as the local disk for storing and accessing locally stored chunks. To this end, the Local Data Manager mainly uses already available functionalities of the Spring Framework, such as Spring Data[7].

Service Transfer Handler The Service Transfer Handler implements the API of the cloud storage nodes and fog storage nodes. Both, Amazon and Google provide Java libraries for their cloud storage services. We use these Java libraries in our implementation of the FogStorage Service to communicate with the cloud storage nodes. As the architecture of the Service Transfer Handler is created with respect to extensibility, further cloud storage nodes can be added. The API of the FogStorage Service itself is implemented to communicate with other FogStorage Service instances. This is needed to store chunks and retrieve chunks as well as to check for free space on other instances of the FogStorage Service. In our implementation, we use the Retrofit library[8] to abstract the REST API calls.

Besides the processes related to data transfer, the Service Transfer Handler implements the authentication process for cloud storage nodes and fog storage nodes. The FogStorage Service uses a YAML property file called `application-auth.yml` to store sensitive data which is needed for authentication. Amazon S3 uses an access key and a secret key for authentication, Google uses a private key file. The access key and the secret key for the Amazon authentication as well as the path to the private key for the Google authentication are part of the `application-auth.yml` file. For authentication on other FogStorage Service instances the own private key is needed which is also part of the `application-auth.yml` file.

Local Database The Spring Framework abstracts the usage of the specific database and allows us to use various different database implementations for the Local Database. The exchange of the underlying database can be done without changing the source code but only by changing the configuration. In our implementation, we use the H2 database[9]. The H2 database has a small footprint and – as it can run in an embedded mode – it is not needed to install another database application on the fog node. In fact, the H2 library is part of the FogStorage Service application which allows to access and store the data on the local disk by the application itself.

[7]https://spring.io/projects/spring-data
[8]https://square.github.io/retrofit/
[9]http://www.h2database.com/

5.1.3 Deployment

In this section, we discuss the deployment of the FogStorage Service. For our implementation, we provide a Maven goal called `package` which builds a `.jar` file. This `.jar` file can be run on every device which has installed the Java Runtime Environment[10].

Being a Spring application, the FogStorage Service offers various configuration possibilities which are available as Spring properties and can be set by adding an `application-prod.yml` properties file next to the `.jar`. Besides the Spring properties, which are documented in the Spring documentation[11], the `application-prod.yml` has some specific properties related to the FogStorage Service. These specific properties are also part of the `application-prod.yml` properties file.

Listing 5.2: FogStorage Service specific properties

```
 1 application:
 2   node-name: FOG_A
 3   nodes-public-config-file: "file:storages_user_info.json"
 4   initial-space-in-mb: 4096
 5   data-path-on-nodes: "./data/"
 6   latency-check-interval-in-seconds: 600
 7   ping-rounds: 4
 8   credentials:
 9     private-key: "private-key"
10     public-key: "public-key"
11   cloud-credentials:
12     aws:
13       access-key: "access-key"
14       secret-key: "secret-key"
15     gcp:
16       project: "fogstorage"
17       private-key-path: "google-private-key-path"
```

In Listing 5.2, the specific properties are shown by using example values. The node-name (Line 2) refers to the unique name of the JSON file containing the information of cloud and fog storage nodes. The nodes-public-config-file property (Line 3) contains the path to this particular file. Furthermore, the specific properties allow to set the initial free space of the fog node for storing chunks (Line 4), as well as the path for storing the chunks (Line 5). The interval for measuring the latency can be configured which also is done by setting a property (Line 6). As the latency is the average value of multiple requests, it is possible to set the amount of requests for calculating this average value, by setting the ping-rounds property (Line 7). The remaining properties refer mainly to the previously mentioned data needed for authentication. Only the own public-key (Line 10)

[10]https://www.java.com/de/download/

[11]https://docs.spring.io/spring-boot/docs/current/reference/html/appendix-application-properties.html

and the project name for the Google Cloud Platform (Line 16) were not mentioned in the description of the `application-auth.yml`. If an `application-auth.yml` properties file is available, the corresponding values in the `application-prod.yml` properties file are overwritten.

The separation of the properties of `application-prod.yml` and `application-auth.yml` has two reasons: First of all, the `application-auth.yml` can be excluded from a source code management system and therefore there is a reduced risk of a leak with sensitive data. Besides that, it is easier to secure the sensitive data of the `application-auth.yml` as it is in an own file. Generally, properties can be secured by encryption with the Jasypt Tool[12].

After placing the generated `.jar`, the `application-auth.yml`, the `application-auth.yml`, and the JSON file with the information of storage nodes and users at the intended places at the fog node, the `.jar` file can be executed using the Java Runtime without any further parameters. As the FogStorage Service is a Spring web server application, the web server specific configuration has to be done according to the documentation[13], such as setting the port and opening the port at the fog node.

5.2 FogStorage Client

In this section, we present implementation details of the FogStorage Client. Based on the related section in Chapter 4, we focus on details which are not part of the design but part of the concrete implementation. We start with a presentation of the used technology stack consisting of the underlying technologies of the Client and the used libraries and tools. After that, we show the implementation details of each component of the FogStorage Client.

5.2.1 Technology Stack & Libraries

The FogStorage Client is, similar to the FogStorage Service, developed with Java 8 and the Spring Framework. For the user interface, we use JavaFX[14]. JavaFX is a framework to offer a graphical user interface on devices which are running the Java Runtime. As dependency management and build tool, we use again Maven.

5.2.2 Implementation Details

According to the defined components of the FogStorage Client, we discuss the different implementation details of each component in the following.

[12] http://www.jasypt.org/

[13] https://docs.spring.io/spring-boot/docs/current/reference/html/howto.html#howto-embedded-web-servers

[14] https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm

User Interface As mentioned, our implementation of the FogStorage Client comes with a graphical user interface. After starting the FogStorage Client the user is prompted for submitting the username and the password. In Figure 5.1, we show the login mask for the user.

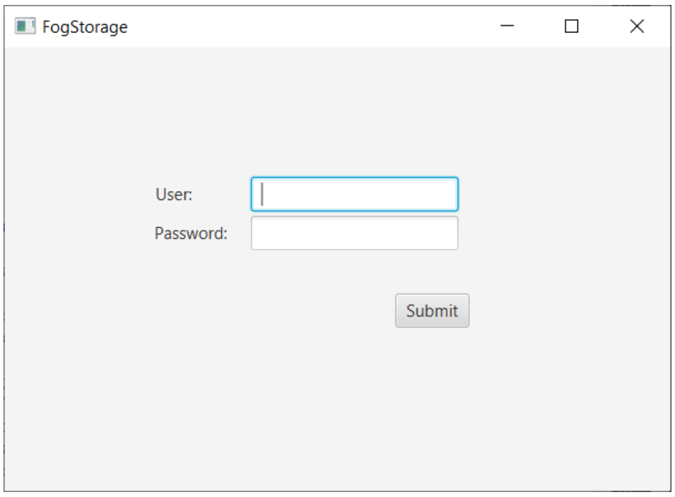
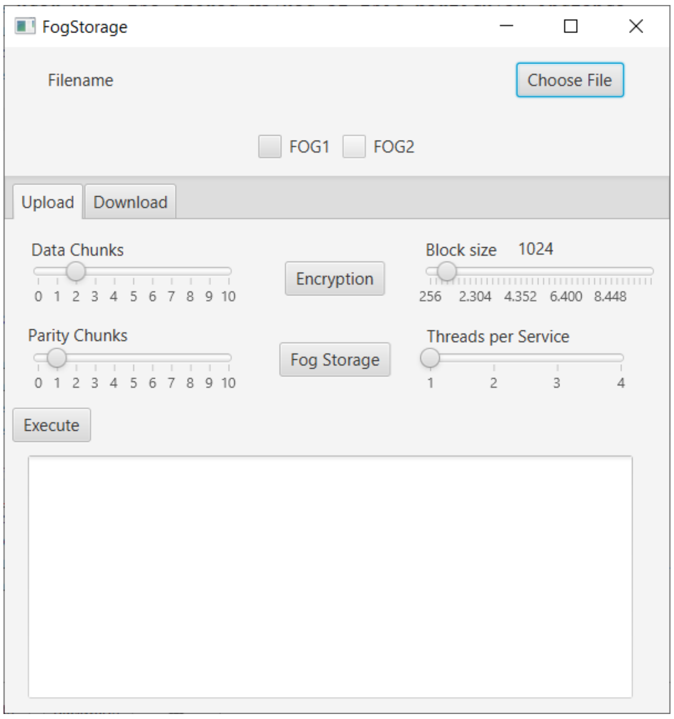


Figure 5.1: User interface: Login mask

After the login, the user is able to upload and download files. In Figure 5.2, we show the graphical user interface for uploading files. By clicking the button 'Choose File', the user can choose a file to upload. After that, the user can choose which available FogStorage Service instances should be participating in the upload process. Besides that, the user can set various upload parameters. We discuss the various details of the parameters at the description of the related component in this section. The parameters are: The number of data and parity chunks to use, the size of one block in kilobytes, and the number of threads one FogStorage Service should use for this particular upload request. Besides that, the user can enable or disable encryption for this particular upload request. Similarly, the user can enable or disable the usage of fog storage for this particular upload request.

After setting the parameters and choosing the file to upload, the user starts the upload by clicking on 'Execute'. After the upload request has finished, the text field below the 'Execute'-Button is filled with information to the request, such as the needed time for processing it.

The user can change to the download mask by clicking on the tab with the label 'Download'. Figure 5.3 depicts the download mask which is very similar to the upload mask. By clicking the button 'Choose File', the user can choose a regeneration-file to download the related original file. The user can then choose which available FogStorage Service instances should be used for downloading the original file. After that, the user can set the number of threads one FogStorage Service should use for this particular download request. Further parameters are not available in Download Mode. Similar to the Upload



Block size 1024

Figure 5.2: User interface: upload mask

Mode, the user can start the download process by clicking on 'Execute'. Again, in the text field below the button, details of the finished download process are shown, such as the needed time for processing.

The graphical user interface is the default interface for a typical user. In our implementation, we provide also a Command Line Interface for advanced users which we describe in Section 5.2.4.

Client Controller As mentioned in Chapter 4, the Client Controller is the component which handles the requests of the user. This mainly consists of preparing the given parameters for further processing. The parameters related to the number of data and parity chunks define the (m, n) erasure coding configuration and are just added to the requests submitted to the FogStorage Service instances. Similarly, the usage of fog storage for a particular request can be enabled or disabled by setting the related parameter. A disabled fog storage refers to the in Section 5.1 mentioned cloud-only Placement Strategy.

As has been previously described, it depends on the concrete implementation how the information of accessible FogStorage Service instances are distributed to the particular FogStorage Clients. In our implementation, the information is stored in a JSON file which is accessible to the FogStorage Client. In Listing 5.3, we show an example of such a file. The file consists of a list containing the available FogStorage Service instances, i.e fog storage nodes. Every fog storage node has a unique name (e.g., Line 4), a URL (e.g.,

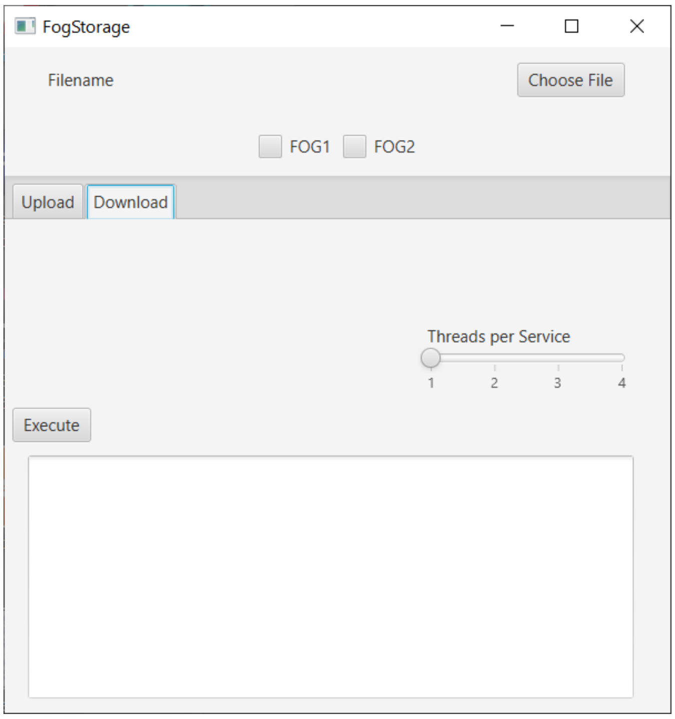


Figure 5.3: User interface: Download mask

Line 5), and a public key (e.g., Line 6). The name of the fog storage node refers to the name of the fog storage node in the JSON file located at the FogStorage Service which contains all available storage nodes (c.f. Section 5.1).

Listing 5.3: Information about FogStorage Service instances

```
 1  {
 2    "nodes": [
 3      {
 4        "name": "FOG_A",
 5        "url": "http://184.172.8.1:8081",
 6        "publicKey": "MIIBIj...hpczHNiwIDAQAB"
 7      },
 8      {
 9        "name": "FOG_B",
10        "url": "http://184.172.8.3:8081",
11        "publicKey": "MIIBIj...A/S+xK8iwIDAQAB"
12      }
13    ]
14  }
```

As described in Chapter 4, the FogStorage Client has to authorize the user at every participating FogStorage Service instance separately. To make this more convenient for the user, the FogStorage Client asks the user for the username and password at start up of the Client and stores these values in a temporal local storage. After the FogStorage Client decides to use a particular FogStorage Service instance for uploading or downloading, the FogStorage Client authorizes the user with the stored values at this particular instance, without the need of involving the user actively. After the authorization is done with a particular instance, the related access token is stored locally, so the authorization has only to be repeated, after the FogStorage Client is restarted or the access token gets invalid.

File Service As described in Chapter 4, the File Service of the FogStorage Client manages the upload and download processes. In Upload Mode, the File Service is responsible for splitting the files into multiple blocks before uploading them. In our implementation, the size of one block in kilobytes can be configured for every upload using the user interface. Furthermore, the File Service utilizes the Client Crypto Handler during upload if encryption is enabled for this particular request. Besides that, the File Service uses the Client Transfer Handler to send upload and download requests to one or multiple FogStorage Service instances. As has been mentioned in the discussion of the user interface, the user can set the number of threads one FogStorage Service should use for one particular upload request. Based on this number of threads, the File Service sends the given amount of requests to one FogStorage Service instance in parallel. For example, if this number of threads is set to one, every FogStorage Service gets just one request and the next one after the processing of the previous request has finished and the response has been received by the File Service. In contrast, if the number of threads is set to three, each FogStorage Service gets three requests at once. As one request has finished and a response is received by the FogStorage Service, the particular FogStorage Service gets another request. This can reduce the whole processing time, as the processing of one request uses one processing thread of the CPU at the FogStorage Service and there are typically further idle threads available.

Client Crypto Handler Similar to the Service Crypto Handler, the Client Crypto Handler uses the default Java cryptography library for symmetric and asymmetric encryption. Besides that, the Client Crypto Handler uses the same operation mode for AES encryption as the Service Crypto Handler and handles the creation and exchange of the initial vector similarly.

Client Transfer Handler The Client Transfer Handler implements the API of the FogStorage Service. This is needed to communicate with the FogStorage instances, i.e., upload and download blocks as well as to authenticate at the FogStorage system. Similarly to the Service Transfer Handler, the Client Transfer Handler uses the Retrofit library to abstract the REST API calls.

5.2.3 Deployment

In this section, we want to shortly discuss the deployment of the FogStorage Client. For our implementation, we provide a Maven goal called `package` which builds a `.jar` file. This `.jar` file can be run on every device which has installed the Java Runtime Environment.

Similar to the FogStorage Service, the FogStorage Client offers various properties which are inherited from a generic Spring application. These Spring properties can be set by adding an `application.yml` properties file next to the `.jar`. Besides the generic Spring properties, also custom properties can be set in this properties file. In our case, we set the fog-nodes-config-file property which is essential for the FogStorage Client. In Listing 5.4, this specific property is shown by using an example value (Line 2). This property contains the path to the JSON file which contains the information of all available FogStorage Service instances. We have already described this JSON file in Section 5.2.2.

Listing 5.4: FogStorage Client specific property

```
1 application:
2   fog-nodes-config-file: "file:fogNodesConfig.json"
```

After placing the generated `.jar`, the `application.yml` and the JSON file with the information of available FogStorage Service instances at the intended place at the user's machine, the `.jar` file can be executed using the Java Runtime. To start the FogStorage Service with the graphical user interface the parameter `-x` has to be added.

5.2.4 Command Line Interface and Scenario Runner

Alternatively to the graphical user interface, our implementation offers a Command Line Interface which enables the user to use the FogStorage Client directly. Every invocation of the FogStorage Client via the Command Line Interface represents one upload or download request. This could be advantageous if the user wants to upload multiple files in a batch process. The FogStorage Client is able to react to various parameters which can be submitted through the command line using the Java Runtime. Listing 5.5 shows the usage of the FogStorage Client with the available parameters.

As can be seen in the description of the usage, there is a Scenario Runner mode and a Key-Generator mode. The Key-Generator Mode generates a private/public key pair and returns both keys as result to the command line. This is just a helper program to get keys for the different components of the system. After generating the keys, this method finishes and stops the further processing. The Scenario Runner mode allows to execute multiple similar requests and is mainly useful for performance tests of the system. As the Scenario Runner mode is needed for the evaluation of our system, we describe this mode with its options in Section 6 in detail.

Listing 5.5: FogStorage Client usage

```
usage: fogStorageClient
 -x Graphical user interface
 -c <arg> Number of data chunks (default: 2)
 -p <arg> Number of parity chunks (default: 1)
 -d <arg> Download file; argument is the regeneration-file
 -u <arg> Upload file; argument is the file to upload
 -e Encryption enabled
 -f Storage in the fog enabled
 -h <arg> FogStorage Service instances to use; Comma-separated list
 -k <arg> Number of kilobytes for one block
 -n <arg> Username to use
 -t <arg> Number of threads per FogStorage Service
 -s <arg> Scenario Runner Mode; the argument is the scenario file
 -g Key-Generator Mode;
          Generates a private/public key pair
          without further processing
```

5.3 Details of the FogStorage Service API

As mentioned in Section 5.1, we present the details of the FogStorage Service API in the following tables. The presentation of the endpoints contains a description of the request, the used HTTP method and URL, the parameters of the request as well as the response of the request. The response part contains the description of the possible HTTP status codes and the corresponding result according to the particular HTTP status code. Tables 5.1 and 5.2 describe the API endpoints for authentication, Tables 5.3 and 5.4 describe the API endpoints for uploading and downloading data blocks, Tables 5.5 and 5.6 describe the API endpoints for uploading and downloading chunks, and Table 5.7 describes the API endpoint for requesting the FogStorage Service instance about the remaining free space for storing further chunks.

Table 5.1: Authentication of users

Description	This endpoint is used for the authentication of users
Request	POST /api/authenticate-user
Parameters:	
login	The login data for the user is added as JSON object to the request. The field *challenge* contains the encrypted random integer. `{"challenge": "string", "encryptedPassword": "string", "encryptedSharedSecret": "string", "username": "string" }`
Response:	
200: OK	If the login process is successful, the response contains the access token and the decrypted random integer in the field *challenge*. `{"challenge": "string", "accessToken": "string" }`
401: Unauthorized	If the login process is not successful, the user gets an error message.

Table 5.2: Authentication of other FogStorage Service instances

Description	This endpoint is used for the authentication of other FogStorage Service instances
Request	POST /api/authenticate-fog
Parameters:	

Login data	The login data for the access requesting FogStorage Service instance is added as JSON object to the request. The field *challenge* contains the encrypted random integer.
	`{"challenge": "string",`
	` "encryptedPassword": "string",`
	` "encryptedSharedSecret": "string",`
	` "username": "string" }`

Response:	

200: OK	If the login process is successful, the response contains the access token and the decrypted random integer in the field *challenge*.
	`{"challenge": "string",`
	` "accessToken": "string" }`

401: Unauthorized	If the login process is not successful, the requesting FogStorage Service instance gets an error message.

Table 5.3: Upload of data blocks

Description	This endpoint is used for uploading data blocks
Request	POST /api/blocks/upload
Parameters:	
dataChunksCount	Request parameter which defines the number of data chunks for the erasure coding configuration to use.
parityChunksCount	Request parameter which defines the number of parity chunks for the erasure coding configuration to use.
useFogAsStorage	Boolean request parameter which defines if the placement strategy should use fog nodes as potential storage nodes.
uploadFile	File with the data block to upload.
Response:	
200: OK	If the upload process is successful, the response contains the encrypted placement-info. `{"encryptedPlacement": "string" }`
401: Unauthorized	If the requesting component is not authorized for this request.
422: Unprocessable Entity	If the upload process is not successful.

Table 5.4: Download of data blocks

Description	This endpoint is used for downloading data blocks
Request	POST /api/blocks/download
Parameters:	
placement	The encrypted placement-info is added as JSON object to the download request. `{"encryptedPlacement": "string" }`
Response:	
200: OK	If the download process is successful, the response contains the block as file.
401: Unauthorized	If the requesting component is not authorized for this request.
422: Unprocessable Entity	If the download process is not successful.

Table 5.5: Upload of chunks

Description	This endpoint is used for uploading chunks
Request	POST /api/chunks
Parameters:	
chunk	The chunk to upload to the FogStorage Service is added as JSON object to the request. `{"bytes": { "content": "string" },` ` "chunk": { "uuid": "string",` `             "size": 0 } }`
Response:	
200: OK	If the upload process is successful.
401: Unauthorized	If the requesting component is not authorized for this request.
422: Unprocessable Entity	If the upload process is not successful.

Table 5.6: Download of chunks

Description	This endpoint is used for downloading the content of chunks
Request	POST /api/chunks/content
Parameters:	
chunk	The chunk whose content should be downloaded from the FogStorage Service is added as JSON object to the request. `{"chunk": { "uuid": "string",` `            "size": 0 } }`
Response:	
200: OK	If the upload process is successful, the content of the chunk is added to the response as JSON object. `{"content": "string" }`
401: Unauthorized	If the requesting component is not authorized for this request.
422: Unprocessable Entity	If the download process is not successful.

Table 5.7: Requesting free space

Description	This endpoint is used for requesting the FogStorage Service about the remaining free space for storing further chunks
Request	/api/chunks/free-space
Response:	
200: OK	If the upload process is successful, the remaining free space is added as integer to the response.
401: Unauthorized	If the requesting component is not authorized for this request.

CHAPTER 6

Evaluation

After we have presented the design and implementation of our FogStorage system in Chapter 4 and 5, we evaluate the system in this chapter. Therefore, we verify the designed and implemented solution for the FogStorage system against the requirements which we have discussed in Chapter 4. As a foundation for this evaluation we use results from experiments done with the implemented FogStorage system. First, we describe the experiments and the prerequisites for the experiments. This description consists of a presentation of the evaluation setup, the used parameters during the execution, and the methodology of the executions. Based on the results of the experiments, we discuss the requirements of the system and how the FogStorage system meets these requirements. We focus on the non-functional requirements as these are crucial to reach the goals of our approach – namely preserving privacy, providing low latency, and offering high availability.

6.1 Experiments

In this section, we discuss experiments with the FogStorage system and the prerequisites for these experiments. The experiments serve as basis for the subsequent evaluation of our approach. First, we present the evaluation setup with the participating components. After that, we discuss the scenarios which represent comparable and repeatable invocations of the FogStorage Client and show the used methodology to execute the experiments.

6.1.1 Evaluation Setup

Figure 6.1 depicts the FogStorage system in our evaluation setup. The cloud layer consists of services offered by cloud storage providers, the fog layer contains two FogStorage Service instances and the user layer consists of the FogStorage Client. The components of the fog layer and user layer are in one local network which is located at TU Wien. We discuss the different layers and components in the following.

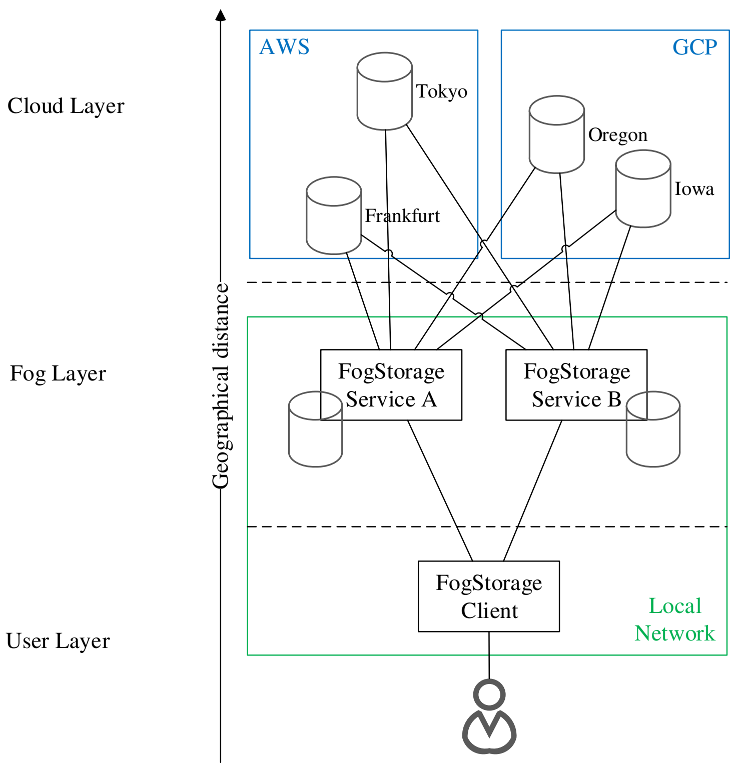


Figure 6.1: Evaluation setup

FogStorage Service

In our evaluation setup, we have two FogStorage Service instances – A and B – which are executed on two virtual machines. Each virtual machine is equipped with 4 virtual CPUs, 4 GB RAM, and 60 GB storage space. These specs are similar to the specs of the Raspberry Pi 4 Model B[1]. A Raspberry Pi is a micro-computer which is used as fog device by different fog-related approaches in their evaluation setups [51, 54, 78]. Therefore, the virtual machine with the given specs is appropriate for simulating a fog device in our evaluation setup.

As we have presented in Chapter 5, the FogStorage Service has some specific properties which can be configured by setting them in the related configuration file. One property is the unique name of the FogStorage Service which we set to `FOG_A` and `FOG_B`, respectively. As we have have described in Chapter 5, the latency of the different cloud storage nodes and fog storage nodes is determined dynamically during the execution of the FogStorage Service. Consequently, the best-suited storage nodes could change

[1]https://www.raspberrypi.org/products/raspberry-pi-4-model-b/

during the process. Two properties are important for the measurement of the latency. First, the `ping-rounds` property which we set to 4 in our evaluation setup. This means that the latency of each cloud or fog storage node is calculated as the average of four requests. As we have identified during pretests, the latency of each storage node does not change significantly. Therefore, an average value of four requests is sufficient. Besides that, the implementation allows to configure the time of the periodical interval to measure the latency. In our evaluation setup, we set this property to 1800 seconds (`latency-check-interval-in-seconds`) which means that every half of an hour the latency of the storage nodes is measured newly. As the latency does not change significantly, we choose a rather large interval to reduce the overhead for the measurement requests.

Furthermore, we set up the authentication of both FogStorage Service instances with the cloud storage providers as well as the authentication of the FogStorage Service instances among each other. We added all the relevant information to the configuration file of the FogStorage Service and to the JSON file with the information of storage nodes and users (`nodes-public-config-file`). Besides that, we added a test user with the username “user” and the bcrypt-hash of “Password” as the password to the JSON file. The detailed description of the needed file structure and content is given in Chapter 5.

Cloud Storage Services

As we have mentioned in Chapter 5, our implementation is able to communicate with the cloud storage services of the Google Cloud Platform (GCP) as well as with the cloud storage services of Amazon Web Services (AWS). In our evaluation setup, we use both of these cloud storage providers – each with two data centers. As defined in Chapter 4, we refer to one data center of a cloud storage provider as cloud storage node. Therefore, we have four cloud storage nodes in sum. The two cloud storage nodes provided by AWS are located in Frankfurt and Tokyo. The two cloud storages provided by GCP are located in Oregon and Iowa.

As mentioned, the latency of the storage nodes is determined dynamically by each FogStorage Service instance separately. These latency values are essential for the Placement Strategy. As we have configured that every half of an hour the latency of the storage nodes is measured newly, it is possible that the measured latency changes during the experiments. Table 6.1 shows the latency values in milliseconds which were measured at the beginning of the experiments by the FogStorage Service instances. According to the described evaluation setup, the latency values are the average of four measurements respectively.

FogStorage Client

In our evaluation setup, we use one FogStorage Client which is executed on a virtual machine. The virtual machine is equipped with 4 virtual CPUs, 4 GB RAM, and 60 GB storage space.

Table 6.1: Initial average latency values of the FogStorage Service instances

FogStorage Service A		FogStorage Service B	
Cloud Storage Node	**Latency (ms)**	**Cloud Storage Node**	**Latency (ms)**
GCP Oregon	184	GCP Oregon	186
GCP Iowa	143	GCP Iowa	147
AWS Frankfurt	25	AWS Frankfurt	26
AWS Tokyo	278	AWS Tokyo	280

As described in Chapter 5, the FogStorage Client needs access to a JSON file which contains the information of all available FogStorage Service instances. According to the description in Chapter 5, we configured the FogStorage Client to be able to connect to both FogStorage Service instances – `FOG_A` and `FOG_B`.

6.1.2 Scenarios

As announced in Chapter 5, the FogStorage Client is equipped with a Scenario Runner Mode which is able to execute multiple requests to the FogStorage Client and to measure the total time of every request. This allows us to define scenarios which consist of comparable and repeatable invocations of the FogStorage Client in different settings. Therefore, we use this mode with its scenarios for our experiments. First, we describe the Scenario Runner Mode of the FogStorage Client. After that, we present the scenarios we are using in our evaluation.

Scenario Runner Mode

The Scenario Runner Mode is a mode of the FogStorage Client. The FogStorage Client can execute multiple requests sequentially which we call a scenario. A scenario is defined by a scenario file. To execute the FogStorage Client in the Scenario Runner Mode, the parameter `-s` with the scenario file as argument has to be submitted. We describe the structure and syntax of the scenario file as well as the characteristics of the Scenario Runner Mode in the following.

The scenario file is a plain text file where every line defines one or multiple invocations of the FogStorage Client via the the Command Line Interface. The Command Line Interface and the parameters are described in Chapter 5. A scenario line – which is an interpreted line of the scenario file – consists of three parts: the number of invocations, the name of the setting, and the command for the invocation of the FogStorage Client via the Command Line Interface. The number of invocations defines how often the related command is executed. The name of the setting is an identifier of the command which allows to interpret the measured results.

The syntax of the scenario line is as follows: The number of the invocations and the name of the setting is separated by an x, the name of the setting and the command is separated by a colon (:). Besides that, comments can be added to the scenario file by prefixing them with a double slash (//). These lines are not interpreted by the Scenario Runner Mode. Furthermore, the Scenario Mode is able to handle variables which are defined and assigned at the beginning of the scenario file. The variables can then be used at every line in the scenario file and are replaced with the assigned value at runtime. Variables are denoted with the identification of the variable between braces.

Listing 6.1 shows an example of a scenario file. The scenario file starts with the variable definitions and assignments from Line 1 to Line 5. Line 7 shows a comment which is ignored by the Scenario Runner Mode. These comment lines should help to structure and understand the file better but do not affect the invocations of the FogStorage Client. In contrast, Line 9 shows a scenario line which is interpreted by the Scenario Runner Mode. This line represents a command which is executed five times. The name of the setting is CE.

Listing 6.1: Scenario File

```
 1 {FILE_2MB}:./data/2MB.zip
 2 {FILE_2MB_FOG}:./data/2MB.zip.fog
 3 {PARITY_CHUNKS}:1
 4 {DATA_CHUNKS}:2
 5 {BLOCK_SIZE}:8192
 6
 7 //Upload File with disabled fog storage and enabled encryption
 8 5xCE:-u {FILE_2MB} -h FOG_A -c {DATA_CHUNKS}
 9      -p {PARITY_CHUNKS} -k {BLOCK_SIZE} -e -n user -t 4
10
11 //Download the previously uploaded files
12 5xCE:-d {FILE_2MB_FOG} -h FOG_A -c {DATA_CHUNKS}
13      -p {PARITY_CHUNKS} -k {BLOCK_SIZE} -e -n user -t 4
14
15 //Upload File with enabled fog storage and enabled encryption
16 5xCFE:-u {FILE_2MB} -h FOG_A -c {DATA_CHUNKS}
17       -p {PARITY_CHUNKS} -k {BLOCK_SIZE} -f -e -n user -t 4
18
19 //Download the previously uploaded files
20 5xCFE:-d {FILE_2MB_FOG} -h FOG_A -c {DATA_CHUNKS}
21       -p {PARITY_CHUNKS} -k {BLOCK_SIZE} -f -e -n user -t 4
```

The Scenario Runner Mode executes the command of every scenario line as often as defined by the number of invocations. The total time of every invocation is measured and added to a csv-file with all results. This csv-file contains five values for every invocation. These values are: The request mode (i.e., upload or download), the setting name, the size of the file in MB, the total time (RTT), and the measured time for encryption and decryption.

Every invocation is executed sequentially, i.e., the second invocation of the command in Line 9 is started after the first invocation has finished. After every invocation of a command, the Scenario Runner Mode waits two seconds before it starts with the next invocation. Similarly, the scenario lines are interpreted sequentially by the Scenario Runner Mode.

Besides that, the Scenario Runner Mode has two characteristics which should ensure the comparability of the results:

1. **Warm-Up:** Before the command of one scenario line is executed the first time, the Scenario Runner Mode executes the same command in a warm-up mode. This means that the command is executed as defined but the result is not added to the result csv-file.

2. **Different Files:** The file changes for every invocation of one scenario line. Before processing the upload request, the Scenario Runner Mode generates a new file with the same size as the original file submitted in the request but with a different content (i.e., random bytes) and a different file name. In detail, the file name is the original file name postfixed with the current number of invocation in this scenario line and with 0 in the warm-up mode. For example, in Line 9 the second invocation of the upload request would be executed with an upload file called `2MB.zip_2`. Consequently, the second invocation of the download request in Line 13 would be executed with a regeneration-file called `2MB.zip_2.fog`.

Evaluation Scenarios

After we presented the details of the Scenario Runner Mode of the FogStorage Client, we discuss the scenarios which we use in our experiments.

For the later evaluation of our approach and the needed comparisons we define different settings in our scenarios. The settings differ in the enabling of storage in the fog and the usage of multiple FogStorage Service instances:

- **CFM**: This setting represents our proposed solution. In this setting, the FogStorage system uses cloud and fog storage nodes with multiple FogStorage Service instances.

- **CF**: Similar to CFM, cloud and fog storage nodes are used by the FogStorage system. But in contrast to CFM only one FogStorage Service instance is used in this setting.

- **C**: In this setting, the FogStorage system uses only cloud storage nodes with only one FogStorage Service instance. This setting should serve as baseline and is inspired by the solution of CORA [14] even though CORA is not intended to be executed on resource-constrained fog nodes. We have presented a detailed discussion about CORA in Chapter 3.

- **CM**: Similar to setting C, the FogStorage system uses only cloud storage nodes in this setting. But in contrast to setting C, this setting intends to use multiple FogStorage Service instances.

Generally, the settings are defined by the parameters for the particular scenario line. One specialty is the parameter for the number of threads per FogStorage Service. As we have described in Chapter 5, this parameter defines how many requests the FogStorage Client sends to one FogStorage Service instance in parallel. According to the described evaluation setup, the FogStorage Client is executed on a virtual machine equipped with 4 virtual CPUs. As we use two FogStorage Service instances at most, we set the number of threads per service to 4 in a setting with one FogStorage Service and to 2 in a setting with multiple FogStorage Service instances. Therefore, the amount of threads the FogStorage Client has to handle is always 4. A higher number of threads per service would not be reasonable as the FogStorage Client would not be able to process this number of threads.

Besides these setting-specific parameters, there are general parameters which are the same for each request in our experiments. In detail, these parameters are the activation of encryption, the block size, and the numbers of data chunks and parity chunks, i.e., the (m, n) erasure coding configuration. We use the following values:

- We enable encryption for all requests in our experiments, but measure the time that is needed for the encryption and decryption. Hence, we can determine at which cost this additional level of privacy comes.

- As mentioned in Chapter 4, the FogStorage Client splits the file into multiple blocks for uploading in order to benefit from the possibility to parallelize the process. Nevertheless, the block size should not be too big because only the process for files with a size below the block size can be parallelized. On the other hand, the block size should not be too small as this would lead to a high overhead for I/O operations, e.g., opening and closing of files. Therefore, we strike a balance between these two requirements and use a block size of 8,192 KB.

- As has been defined in Section 4.3.1, the chosen erasure coding configuration (m, n) is only valid for our Placement Strategy if $m \geq 2$ and $m < n$. In our experiments, we use a (2,3) erasure coding configuration. Hence, the number of data chunks is 2 (m) and the number of parity chunks is 1 $(n - m)$. This means that any 2 chunks out of 3 are enough to regenerate the original block. The used (2,3) erasure coding configuration is the smallest possible configuration according to the constraints defined in Section 4.3.1. Higher numbers for the (m, n) parameters could also be used but since the number of nodes is limited, we chose a small erasure coding configuration. As in all settings the same erasure coding configuration is used, the results are comparable.

In our experiments, we use a set of files from 2 MB to 1000 MB. In detail, we execute our scenarios with the following file sizes which correspond approximately to the given real world examples.

- 2 MB: E-Book
- 50 MB: MP3 Album
- 250 MB: Short video
- 750 MB: CD Rom
- 1000 MB: Longer video in HD quality

Based on these settings, general parameters, and files, we create two scenarios. Scenario 1 executes the following procedure for every file:

- 5 uploads and 5 downloads using setting C with FogStorage Service A
- 5 uploads and 5 downloads using setting CM with FogStorage Service A and FogStorage Service B
- 5 uploads and 5 downloads using setting CF with FogStorage Service A
- 5 uploads and 5 downloads using setting CFM with FogStorage Service A and FogStorage Service B

Scenario 2 executes the following procedure for every file:

- 5 uploads and 5 downloads using setting C with FogStorage Service B
- 5 uploads and 5 downloads using setting CM with FogStorage Service B and FogStorage Service A
- 5 uploads and 5 downloads using setting CF with FogStorage Service B
- 5 uploads and 5 downloads using setting CFM with FogStorage Service B and FogStorage Service A

Therefore, we have two scenarios which each execute the FogStorage Client five times for every of the four settings and for each of the five files. In our experiments, we executed both scenarios two times. That means that every setting is executed for each file 20 times – for upload as well as for download.

6.2 Evaluation Results

In this section, we evaluate our solution of the designed and implemented FogStorage system. To this end, we use the approach of software verification which is the process of confirming that the implemented software meets the defined requirements [58]. Therefore, we discuss the functional and non-functional requirements of the FogStorage system – which we have defined in Chapter 4 – point by point and show how our implemented FogStorage system meets these requirements. Partly, we verify the requirements by using the results of the experiments. As the experimental results can not verify every requirement, we present an analysis of the designed and implemented solution for the remaining requirements.

As defined in Chapter 4, the functional requirements contain the functionality of uploading and downloading files. These requirements are met as the experiments show that this functionality is given. Besides that, we implemented unit tests which are executed automatically before every deployment. Unit testing describes the method of testing software code at its smallest functional point [79]. This means that different classes and methods of the code are already tested during the development process. Therefore, errors can be detected very early and the risk of introducing new errors with source code changes can be minimized.

As the functional requirements are met, we discuss the non-functional requirements in the following. The non-functional requirements are essential for our implementation to reach the goals of preserving privacy, providing low latency, and offering high availability. The upcoming discussion follows the structure of the requirement definition in Section 4.1.2.

6.2.1 Performance Efficiency

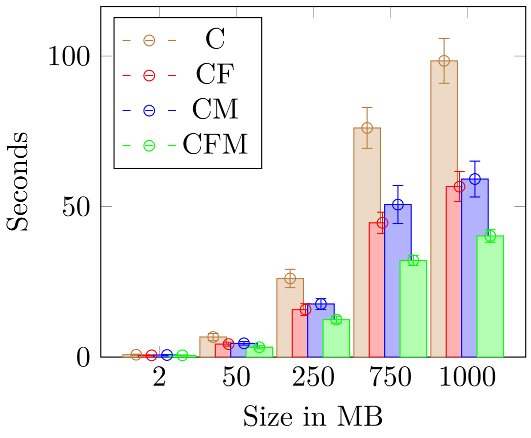
The requirements related to performance efficiency consist of the more specific requirements regarding time behavior and resource utilization which we discuss in the following.

Time Behavior

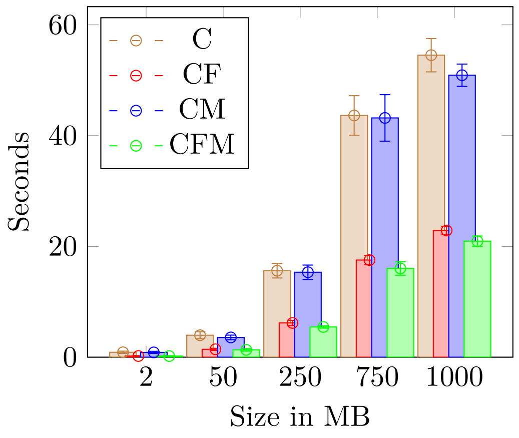
The requirement related to time behavior is defined as offering lower latency than a cloud-only solution. We verify this particular requirement by analyzing the results of the previously described experiments.

We compare the results of the experiments regarding the different settings and file sizes. Figure 6.2 shows a visual representation of the results in an overall comparison. Concretely, Figure 6.2a visualizes the results for uploading and Figure 6.2b shows the results for downloading. The figures show for each setting and each file size an error bar with the average latency (RTT) of the 20 experiments. The standard deviation of the experiments is represented by the error. A detailed overview of the used values in these and the other following figures is shown in Appendix B.

As can be seen in Figure 6.2a the settings CF and CFM which are our proposed solution perform better than C and CM during uploading. Specifically, the setting C has very



(a) Upload



(b) Download

Figure 6.2: Overall comparison

high latency values compared to the other settings. Using a second FogStorage Service instance with the setting CM, reduces the average latency values by about 30% whereas the absolute and relative difference between C and CM increases with higher file sizes. The reason therefore is that the one FogStorage Service instance needs time for the erasure coding, applying the Placement Strategy, and uploading the chunks to the cloud storage nodes. With a second FogStorage Service instance – which is used in the setting CM – this workload can be split. Although the setting CF uses only one FogStorage Service instance, it performs better than the setting CM. Since it uses fog storage nodes the setting CF is able to compensate the needed time for the processing workload. In sum, CF reduces the average latency values by about 9% compared to CM. Using a second FogStorage Service instance with the setting CFM, the average latency values are reduced by about 20% compared to CF whereas the absolute and relative difference between CF and CFM increases with higher file sizes.

Similar to the results related to uploading, Figure 6.2b shows the latency values for the different settings in download mode. All in all, the latency values for all settings are about 40% lower in download mode than in upload mode. A reason therefore is that the processing workload in download mode is not that high as in upload mode. Besides that, a higher amount of data has to be uploaded than downloaded as in our evaluation setup a (2,3) erasure coding configuration is used. It can be seen that our proposed solution – with the settings CF and CFM – perform better than the settings C and CM. Again, the setting C has the highest latency values. Using a second FogStorage Service instance with the setting CM reduces the latency values only by about 4%. Due to the reduced workload, the additional FogStorage Service instance has not that effect as in upload mode. However, the setting CF reduces the average latency values by about remarkable 62% compared to CM. The difference between CM and CF is much higher in download mode than in upload mode. A reason therefore is that during downloading the setting CF gets half of the data from the near fog storage nodes while during uploading the setting CF has to upload two thirds of the data to cloud storage nodes. Besides that, while in upload mode the one FogStorage Service instance with setting CF has to compensate the high workload first, the benefit of using storage nodes affects the total latency in download mode more. Similar to the relation of C and CM, a second FogStorage instance with the setting CFM does not have a huge effect. In detail, the setting CFM reduces the average latency values about 8% compared to setting CF.

Figure 6.3 shows box plots for a more detailed comparison of the different settings in upload mode. Similarly, Figure 6.4 presents box plots for a comparison of the different settings in download mode. For each file size and each setting, the box plots show the values of the median, the upper and lower quartile, the upper and lower whisker as well as the outliers. The values for the median, the upper quartile and the lower quartile are presented in Appendix B. The upper whisker is located at the highest value, but at most at 1.5 times the inter-quartile range from the upper quartile. The lower whisker represents the lowest value, but at least 1.5 times the inter-quartile range from the lower quartile. The inter-quartile range is defined as the difference of the upper quartile and the lower quartile. Values which are not in the range of both whiskers are called outliers and represented as a dot [80]. It has to be considered that the representation of the measurements differ for each file size in these figures as they aim to show the differences of the settings in detail.

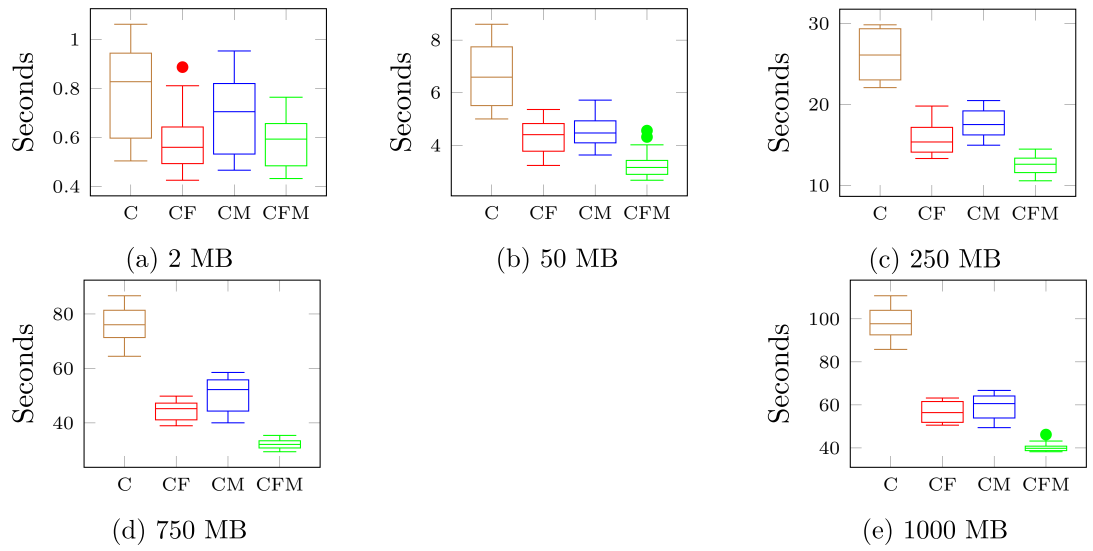


Figure 6.3: Upload: detailed comparison

As can be seen in Figure 6.3 and in Figure 6.4, the distance from the whiskers to the median for the settings using fog storage nodes – namely CF and CFM – is generally shorter than for the settings using only cloud storage nodes (C and CM). A shorter distance from the whisker to the median means that the measured values deviate less

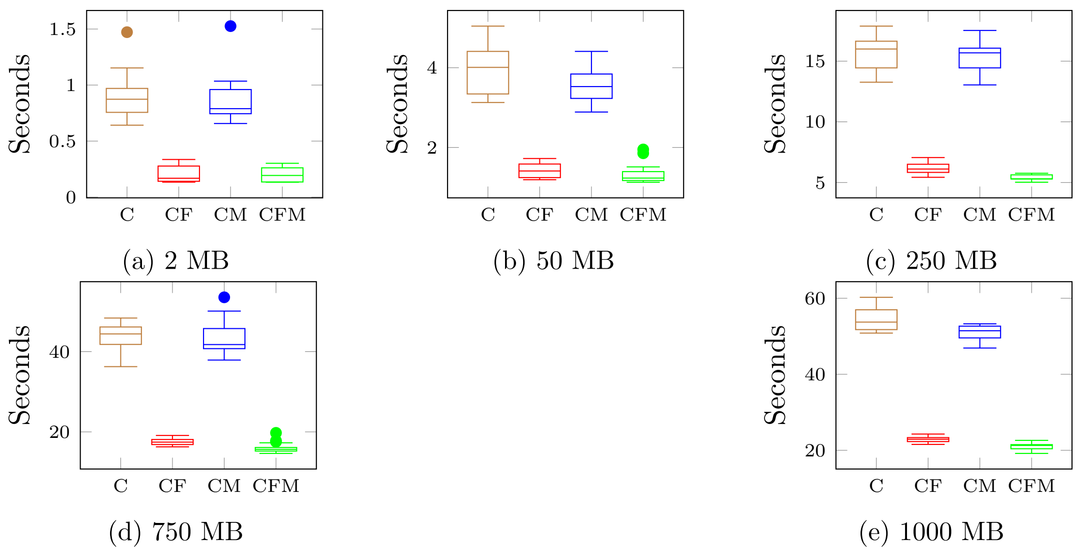


Figure 6.4: Download: detailed comparison

from the median. A reason therefore is that the latency values for cloud storage nodes vary more than for near fog storage nodes. The lower deviation of the settings CF and CFM is already visible in Figure 6.3 showing the values for uploading, but the difference is more remarkable in Figure 6.4 presenting the values for downloading. The difference between uploading and downloading results from the amount of interactions with cloud storage nodes. As a (2,3) erasure coding configuration is chosen, in upload mode the settings CF and CFM have two thirds of their interactions with cloud storage nodes but in download mode they have only a half of their interactions with cloud storage nodes. Therefore, the deviation of the measured latency values is lower during downloading than during uploading.

The requirement for time behavior is defined as offering lower latency than a cloud-only solution. As we have seen at analyzing the results of the experiments, the settings which use fog storage nodes – CF and CFM – have lower latency values in all combinations than the settings without fog storage nodes. Therefore, this specific requirement is met.

Resource Utilization

As defined by the requirement related to resource utilization, the system should offer the same level of data redundancy as with full replication but should need less storage space.

As we have discussed in Section 2.6, using erasure coding can reduce the storage overhead. The FogStorage system uses erasure coding for redundant storage which can be configured by the parameter for the number of data chunks and the parameter for the number of parity chunks. As we have mentioned, we set the data chunk parameter to 2 (m) and the parity chunk parameter to 1 ($n - m$) in our experiments. This means that 3 chunks are distributed to three different storage nodes. Therefore, we could tolerate one storage node failure in our evaluation setup. Considering full replication, one storage node failure can be tolerated if two storage nodes are used and the data chunk is copied completely to both storage nodes. Therefore, it can be said that the erasure coding configuration for a full replication is a (1,2) configuration. That means that the number of data chunks is set to 1 (m) and the parity chunk parameter is also set to 1 ($n - m$). As the storage overhead is calculated by $\frac{n-m}{m}$, we have a storage overhead of 0.5 in our evaluation setup. The storage overhead with full replication would be 1. Therefore, our system offers the same level of redundancy but with less storage space needed which leads to the fulfillment of this specific requirement.

Besides that, we analyzed the resource utilization of one FogStorage Service executed on one virtual machine. After start up, the FogStorage Service does need less than 1% of the available CPU resources in idle mode. As expected, the FogStorage Service takes up to 100% of the available CPU resources while processing upload and download requests. As our approach aims to reduce the overall processing time, this is a desired behavior. Besides that, the memory usage of the FogStorage Service is at most about 25% of the available RAM. As can be seen, the execution of the FogStorage Service on the virtual machines is using the available resources efficiently without bringing the virtual machines to their limits. Therefore, our approach is well suited for an execution in the area of the resource-constrained devices in the fog.

6.2.2 Compatibility

As defined in Chapter 4, there is one specific requirement for compatibility which we discuss in the following.

Interoperability

The requirement of interoperability aims at using a well-known standard for the interfaces of the components and a published and accessible description of these interfaces.

In Chapter 5, we have described the API of the FogStorage Service in detail. The API is used by the FogStorage Client as well as by other instances of the FogStorage Service. The detailed description of every API endpoint is contained in Section 5.3. Besides that, we use the standardized REST protocol for these API endpoint which represents a well-known standard. The FogStorage Client offers a graphical user interface as well as a Command Line Interface. Both interfaces are described in detail in Chapter 5.

The implemented components and the users of the system can communicate by using these defined and described interfaces. Besides that, the interfaces can be used by other new implementations. For example, an exchange of the FogStorage Client with an implementation in another programming language would work seamlessly as the new implementation knows which parameters are needed for a request and what the response should contain.

As the detailed description is part of this work and the well-know standards are used, this specific requirement is met.

6.2.3 Security

The requirements related to security are essential for our approach to reach the goal of preserving privacy. We discuss the specific requirements of confidentiality, integrity, and authenticity as well as our implementation related to these points in the following.

Confidentiality

As defined by the requirement related to confidentiality, only users should have access to the original content of the file or a part of the original content, if these users have also access to the corresponding regeneration-file.

As we have described in Chapter 4, the implemented Placement Strategy ensures that no single storage node can regenerate the original file. Every storage node has at most one chunk of the original block but this chunk could be read by everyone who has access to the file. Therefore, the FogStorage Client encrypts the original block before uploading it to the FogStorage Service. The used secret key is part of the regeneration-file. Therefore, the original file can only be read by the user who has access to the regeneration-file. Even if an attacker gets access to a chunk, the content of the chunk would not be readable and therefore useless for the attacker. Hence, our implementation meets the requirement related to confidentiality.

Of course, this added layer of privacy does not come for free. Table 6.2 shows the average time needed for encryption and decryption for the various file sizes. Besides that, the table shows the standard deviation. As the encryption and decryption happens at the FogStorage Client, the time needed for encryption and decryption is not affected by the chosen setting. Hence, the values are taken from all experiments independent from the chosen setting. As the encryption and decryption time is part of the total Round Trip Time (RTT), the encryption and decryption time is included in the values which we have presented in Figure 6.2a and Figure 6.2b. Figure 6.5 visualizes the values for encryption and decryption as an error bar where the standard deviation is represented by the error.

Nevertheless, our implementation of the FogStorage system allows to deactivate the encryption which would reduce the total time for uploading and downloading. Without encryption, the related chunks of the original block would be stored in clear text at the storage nodes. If an attacker gets access to a storage node, the attacker could steal this information. This is more serious with chunks related to text files than with chunks related to pictures or videos. A small part of a picture or of a video does not indicate how the original file looks – especially if it is not clear which format the original file has. Furthermore, if the attacker gets access to multiple chunks or even all ever stored chunks, it is hard to merge these chunks to blocks which in turn can be combined to the original files. The chunks themselves do not have any information how they are related to each other. On the other hand, chunks related to text files contain human-readable content.

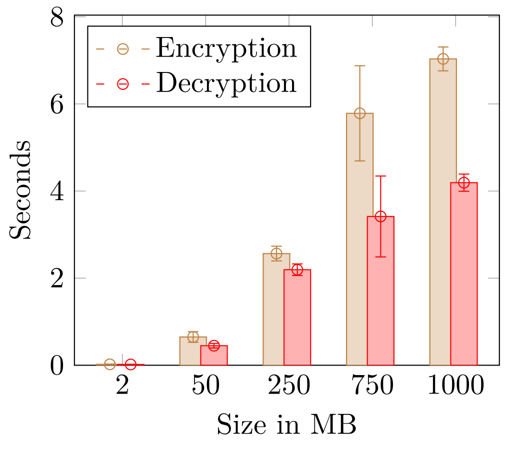


Figure 6.5: Encryption and Decryption

Table 6.2: Encryption and decryption Values

Size	Encryption Average	Encryption σ	Decryption Average	Decryption σ
2	0.021	0.008	0.018	0.005
50	0.649	0.122	0.449	0.053
250	2.564	0.169	2.194	0.133
750	5.784	1.093	3.416	0.929
1000	7.034	0.274	4.192	0.197

Small parts could already contain sensitive information. Furthermore, the attacker could combine the chunks based on the content to get the original block which is easier than with non human-readable content. The user of the system is able to decide for every single file if the additional layer of privacy is worth the higher total time for uploading and downloading.

Integrity

The requirement related to integrity is defined as ensuring that the stored data is not modified – unintentionally or intended by an attacker. In the context of the FogStorage system that would mean that a change to the content of a stored block has to be detected before regenerating the original file. Otherwise, the user could receive a file with different content in download mode than the user has uploaded.

Due to the characteristics of the FogStorage system, it is hard to change the content of one or multiple stored chunks such that the content of the original file is altered in a meaningful way. One chunk is part of one block which in turn is part of the original file. Besides that, each block is encrypted. A change to one chunk would have to be made in a way that neither the merge of the chunks nor the decryption of the block are able to detect this change. Assuming an attacker would manage to overcome these checks by altering one or multiple blocks appropriately, the content of the regenerated file would be garbage and not readable as the original content can not be known to the attacker.

Therefore, a change to one or multiple chunks such that the resulting content is still meaningful is very difficult to achieve. Nevertheless, our implementation provides an additional layer for improving integrity by checking the checksum for each block. As described in Chapter 5, we use a MD5 hash function to calculate the checksum in upload mode before erasure coding the block into multiple chunks. This checksum is part of the the regeneration-file. In download mode – after merging the chunks to one block –

the checksum of the block is calculated again and compared to the value stored in the regeneration file. If the check fails, the system stops the further processing. As the checksum acts only as an additional layer of integrity, we put up with the weakness of the MD5 hash function concerning collision-attacks [81].

As described, a change to the stored data such that the content of the original file changes is not feasible. Therefore, the specific requirement of integrity is met.

Authenticity

As defined by the requirement of authenticity, only a defined set of users and components can participate in the related processes. The system has to ensure that only these users and components are able to use the system and unknown users and components have to be excluded from participating in the related processes.

In Section 4.4, we have described the authentication process very detailed. The FogStorage Service instances have the information of the registered users and of the other FogStorage Service instances. The authentication process verifies that communication is only done with trusted participants. Furthermore, the FogStorage Client knows which FogStorage Service instances can be used for uploading and downloading. Similarly, the authentication process verifies that the communication is only done with trusted FogStorage Service instances. Therefore, the specific requirement of authenticity is met.

6.2.4 Portability

The requirements related to portability consist of the more specific requirements regarding installability and replaceability which we discuss in the following.

Installability

As we have defined in Section 4.1, installability refers to the characteristic of the system that the components can be installed and uninstalled on the most popular operating systems. As we have described in Chapter 5, our implementation of the FogStorage system – the FogStorage Service as well as the FogStorage Client – is implemented in Java. Java runs on concrete implementations of the Java Virtual Machine (JVM). The JVM is an abstract virtual machine which allows to execute Java applications independently from hardware and operating systems [82]. There are many different concrete implementations of the JVM for different operating systems [83]. Therefore, the requirement of installability is met.

Replaceability

As defined by the requirement of replaceability, it should be possible to replace a failing component without knowing details of the failing component.

In our evaluation setup, we have two FogStorage Service instances – A and B. If one FogStorage Service fails, the other one can help out. For instance, the user uploads a big file with the setting CFM. The FogStorage Client would split the file into multiple blocks and distribute them to both FogStorage Service instances for uploading. Assuming that during the process FogStorage Service A fails, FogStorage Service B would process the received requests regardless of that. The FogStorage Client would detect that FogStorage Service A is not available anymore and would distribute the remaining blocks to FogStorage Service B only. As the FogStorage Client does not get a successful response from FogStorage Service A for a particular amount of the blocks, these blocks are again distributed to FogStorage Service B. The file would have been uploaded successfully despite the failure of one FogStorage Service instance.

The FogStorage Service instances can replace each other – even during an upload or download process. Therefore, the specific requirement of replaceability is met.

6.2.5 Reliability

The requirements related to reliability are essential for our approach as one main goal is to offer high availability. Besides availability, recoverability is part of the section related to reliability. We discuss both specific requirements in the following.

Availability

As defined in Chapter 4 by describing the requirement of availability, the system should be able to regenerate the original file despite the failure of one storage component. Besides that, the original file should also be accessible if the FogStorage Service instance which is used to download the file is a different one to the instance which was used to upload the file. For example, a user could upload a file at home where one FogStorage Service – and therefore a fog storage node – runs in the user's local network. During upload, a particular amount of chunks would be placed on this fog storage node. The user could change to another place where the services of the local network are not accessible but should also have access to the file.

We have already discussed the used erasure coding configuration at the requirement related to resource utilization. As we have described, our evaluation setup is configured to distribute three chunks to three different storage nodes, but only needs two of these chunks to regenerate the original file, i.e., we use a (2,3) erasure coding configuration. Figure 6.6 shows at which storage nodes the chunks of our experiments are stored. Generally, the used storage nodes are the ones with the least latency values according to Table 6.1. There is only one exception: In setting CFM, the FogStorage Service decided to store a part of the chunks at the GCP Oregon storage node. The reason therefore is that the latency is determined dynamically during the execution of the FogStorage Service. Hence, the FogStorage Service classifies the GCP Oregon storage better than the GCP Iowa storage node related to latency for a particular time span.

As expected, the settings CF and CFM use fog storage nodes whereas C and CM only use cloud storage nodes. Independent from the chosen setting, it is possible to regenerate the original file if one storage node fails. The settings CF and CFM could also tolerate

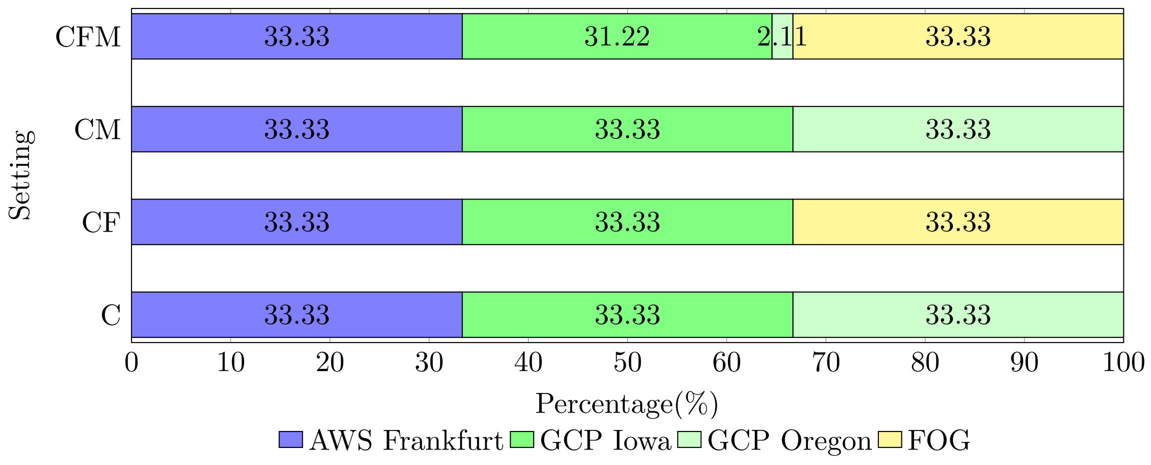


Figure 6.6: Distribution of data to the different storage nodes

that all storage nodes of one particular cloud storage provider are not accessible. In contrast, the settings C and CM would not be able to tolerate an outage of the GCP.

As described in Chapter 4, the Placement Strategy chooses at least m cloud storage nodes. Regarding our evaluation setup, this means that at least two cloud storage nodes and at most one fog storage node are chosen. This constraint ensures that the original file can be regenerated also if the fog storage node is not temporarily accessible – e.g. due to a change of the local network. Therefore, the requirement of availability is met.

Recoverability

The requirement of recoverability refers to the characteristic of a system being able to recover from a failure of one component and to add new components of this kind.

As we have verified the requirement of replaceability, we have already described how the system is able to replace failed components and how the requests are handled independently. As it could be the case that the failed component is damaged and can not be restarted, new FogStorage Service instances could be added. As described in Chapter 5, the deployment of the FogStorage Service consists of executing the `.jar`-file, configuring the properties-files, and putting the JSON file with the information of storage nodes and users at the intended place. After deployment, the information concerning this new FogStorage Service has to be updated at every FogStorage Service and every FogStorage Client.

As new FogStorage Service instances can be deployed without the need of a state exchange with other failing FogStorage Service instances, the requirement of recoverability is met.

In this chapter, we presented the details of the evaluation setup and a description of the experiments and their execution. Based on the results of the experiments and by analysis of the characteristics of the designed and implemented system, we verified the FogStorage system against the requirements described in Chapter 4. As all requirements could be verified, it can be stated the FogStorage system reaches the goals of preserving privacy, providing low latency, and offering high availability.

CHAPTER 7

Conclusion and Future Work

In this final chapter, we summarize the work done in this thesis and give an outlook for potential further extensions of this work. We start with a conclusion containing the summary of the main parts of our approach in Section 7.1 and show possible future work based on our approach afterwards in Section 7.2.

7.1 Conclusion

Cloud storage services offer various advantages to their users but still have limitations related to privacy, latency, and availability. In this thesis, we addressed these limitations by presenting a storage solution which uses resources at the edge of the network according to the fog computing paradigm. Before we started with the design and implementation of this solution, we first analyzed related work and presented a detailed description of the most relevant approaches related to cloud-based and fog-based storage. We discussed the different features of these approaches and compared them to our proposed solution.

After that, a definition of the requirements of a fog-based storage system with the main goals of preserving privacy, providing low latency, and offering high availability was given. Based on these requirements, we designed the FogStorage system which consists of two parts, namely the FogStorage Service and the FogStorage Client. The FogStorage Service is intended to be executed on fog nodes whereas the FogStorage Client is meant to be run on the user's machine. A core part of the FogStorage Service is the Placement Manager which implements the Placement Strategy. As the Placement Strategy is responsible for finding a suitable placement for the uploaded data, it is essential for fulfilling the previously defined requirements. First, the Placement Strategy ensures that the originally uploaded data cannot be read or analyzed by anyone but the owner of the data. Second, the Placement Strategy distributes the data in a way that the overall latency is reduced compared to cloud-only solutions. Third, the Placement Strategy increases availability as despite of a failure of one storage component the data would still be accessible. In

Chapter 5, we presented the concrete implementation of the FogStorage system based on the previously established design. Both components of the FogStorage system – the FogStorage Service and the FogStorage Client – are implemented in Java. We discussed the different implementation details and described the deployment of both components. Finally, we presented the evaluation of our approach in Chapter 6. With the implemented Scenario Runner Mode, we were able to execute different experiments and compare our approach to cloud-only solutions.

Based on the experiments which we executed using multiple distributed cloud and fog storage nodes, we conclude that various aspects of cloud storage can benefit from using fog computing. Specifically, the latency can be reduced considerably. Depending on the file size, the experiments show that the latency can be reduced by 15-42% in Upload Mode and even by 58-76% in Download Mode. Besides that, the experiments in our evaluation setup have shown that by using fog nodes our implementation is able to tolerate even an outage of one cloud storage provider with all its cloud storage nodes. Furthermore, with an analysis of the used mechanisms we have shown that our implementation of the FogStorage system fulfills all requirements defined in Chapter 4. Therefore, we conclude that our approach is able to preserve privacy, provide low latency, and offer high availability.

7.2 Future Work

In this thesis, a fog-based storage system with the main goals of preserving privacy, providing low latency, and offering high availability was designed, implemented, and evaluated. The work done in this thesis offers a foundation for potential further work. In the following, we discuss possible subjects.

Cost-Efficient Placement We implemented the FogStorage Service and the related Placement Strategy with aiming at preserving privacy, providing low latency, and offering high availability. This implemented Placement Strategy could be optimized by extending it with a cost-efficient placement. As the pricing scheme for the cloud storage services varies from cloud storage provider to cloud storage provider, the selection of a particular cloud storage service has an impact on the costs for storing the data [14]. Therefore, adapting the Placement Strategy in order to consider the pricing scheme of the different cloud storage services could lead to lower overall costs while preserving privacy, providing low latency, and offering high availability.

Dynamic Configuration The FogStorage Service requires information about the accessible cloud storage services and other executing FogStorage Service instances. Similarly, the FogStorage Client needs information about accessible FogStorage Service instances. In our implementation, we provided this information in a JSON file which is part of the deployment. This JSON file can be adapted during runtime but has to be changed manually. The information of this JSON file could be distributed automatically. An approach would be that one central component could store this information and the participating components could retrieve it from this central component – after authentication. This would be a more convenient approach as the information would be updated at every participating component without manual work. Besides that, this would allow to implement some kind of registration process for new FogStorage Service instances or FogStorage Clients.

Intra-fog chunk exchange The Placement Strategy decides which chunks should be stored in the fog and which fog storage nodes should store these chunks. After that, these chunks stay at the selected fog storage nodes. A potential extension of the FogStorage system could be a mechanism to exchange chunks between fog storage nodes over time. One FogStorage Service instance could exchange the locally stored chunks with another FogStorage Service instance or multiple other FogStorage Service instances. This data exchange could be part of a backup strategy for the chunks which are stored in the fog. This could increase the availability of the stored data in the FogStorage system even more. Besides that, the data exchange could help to establish a cache mechanism or act as a way to migrate data. It could be the case that the user has changed the location and has now access to other fog storage nodes than in Upload Mode. Temporarily, the system could decide to copy the chunks to the now accessible fog storage nodes to establish a cache mechanism. If the system detects that the new location is permanent, the system could decide to migrate the data from the original fog storage node to the now accessible fog storage node.

APPENDIX A

Notation of the Placement Strategy

Table A.1: Notation of the Placement Strategy – used in Section 4.3.1

Symbol	**Description**
n	n is the amount of created chunks for one block to upload.
m	m is the amount of needed chunks for regenerating one particular block.
b	b is the size of one chunk in bytes.
$c \in C = \{c_1, c_2, \ldots\}$	C is the set of accessible cloud storage nodes and c is a specific accessible cloud storage node.
$f \in F = \{f_1, f_2, \ldots\}$	F is the set of accessible fog storage nodes and f is a specific accessible fog storage node.
$s \in S = \{s_1, s_2, \ldots\}$	S is the set of accessible storage nodes and s is a specific accessible storage node, i.e., $S = C \cup F$.
l_{s_i}	The value l_{s_i} describes the latency between the fog node executing the Placement Strategy and the accessible storage node s_i.
$d \in D = \{d_1, d_2, \ldots\}$	D is the set of all chunks of one particular block and d is a specific chunk of one particular block, i.e., $\|D\| = n$.
$\tilde{C} \subset C$	$\tilde{C}$ is the subset of C which consists of the selected cloud storage nodes for the placement.
$\tilde{F} \subset F$	$\tilde{F}$ is the subset of F which consists of the selected fog storage nodes for the placement.
$\tilde{S} \subset S$	$\tilde{S}$ is the subset of S which consists of the selected storage nodes for the placement, i.e., $\|\tilde{S}\| = \|D\| = n$ and $\tilde{S} = \tilde{C} \cup \tilde{F}$.
$S' = \{s'_1, s'_2, \ldots\}$	S' is the ordered set of accessible storage nodes according to the latency in ascending order, i.e., $l_{s'_i} \leq l_{s'_{i+1}}$.
$S^* \subset \tilde{S}$	S^* is the subset of $\tilde{S}$ which consists of the best-suited storage nodes for regenerating the particular block, i.e., $\|S^*\| = m$.
x	x is used in Algorithm 4.1 and Algorithm 4.2 as temporary helper variable.

APPENDIX B

Measured Latency during the Experiments

Table B.1: Values based on the measured latency during the experiments for uploads

Size	**Approach**	**Average**	σ	Q_1	Q_2	Q_3
2	C	0.784	0.186	0.597	0.827	0.944
	CF	0.587	0.127	0.493	0.559	0.642
	CM	0.688	0.149	0.532	0.705	0.820
	CFM	0.582	0.106	0.484	0.593	0.656
50	C	6.666	1.210	5.512	6.592	7.745
	CF	4.315	0.663	3.777	4.407	4.829
	CM	4.577	0.589	4.096	4.470	4.932
	CFM	3.271	0.528	2.897	3.160	3.427
250	C	26.137	3.057	23.015	26.087	29.329
	CF	15.771	1.929	14.100	15.354	17.157
	CM	17.639	1.779	16.225	17.504	19.190
	CFM	12.449	1.157	11.576	12.605	13.358
750	C	76.122	6.761	71.341	76.043	81.349
	CF	44.588	3.574	41.120	45.225	47.254
	CM	50.649	6.343	44.338	52.227	55.804
	CFM	32.122	1.626	30.789	32.093	33.432
1000	C	98.405	7.448	92.544	97.713	103.937
	CF	56.617	4.997	51.877	56.411	61.552
	CM	59.158	5.965	53.908	60.591	64.132
	CFM	40.237	2.045	38.748	39.822	40.814

Table B.2: Values based on the measured latency during the experiments for downloads

Size	Approach	Average	σ	Q_1	Q_2	Q_3
2	C	0.887	0.200	0.757	0.874	0.970
	CF	0.207	0.075	0.142	0.169	0.278
	CM	0.867	0.193	0.745	0.790	0.960
	CFM	0.202	0.063	0.136	0.194	0.262
50	C	3.963	0.627	3.343	4.009	4.410
	CF	1.422	0.185	1.245	1.411	1.585
	CM	3.569	0.414	3.231	3.528	3.843
	CFM	1.319	0.229	1.172	1.233	1.395
250	C	15.611	1.310	14.447	16.001	16.644
	CF	6.182	0.480	5.833	6.110	6.503
	CM	15.337	1.294	14.447	15.686	16.069
	CFM	5.451	0.209	5.296	5.296	5.631
750	C	43.636	3.574	41.791	44.403	46.135
	CF	17.536	0.830	16.848	17.447	18.104
	CM	43.198	4.199	40.740	41.754	45.742
	CFM	16.017	1.217	15.224	15.645	16.112
1000	C	54.523	3.009	51.754	53.731	56.969
	CF	22.868	0.697	22.302	22.908	23.341
	CM	50.901	2.015	49.566	51.448	52.661
	CFM	20.943	0.927	20.411	21.281	21.492

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] P. Waibel, J. Matt, C. Hochreiner, O. Skarlat, R. Hans, and S. Schulte, "Cost-optimized redundant data storage in the cloud," *Service Oriented Computing and Applications*, vol. 11, no. 4, pp. 411–426, 2017.

[2] R. Pottier and J. Menaud, "Trustydrive, a multi-cloud storage service that protects your privacy," in *9th IEEE International Conference on Cloud Computing(CLOUD 2016)*, pp. 937–940, 2016.

[3] P. Mell and T. Grance, "The nist definition of cloud computing," *Communications of the ACM*, vol. 53, no. 6, p. 50, 2010.

[4] A. Botta, W. de Donato, V. Persico, and A. Pescapè, "Integration of cloud computing and internet of things: A survey," *Future Generation Comp. Syst.*, vol. 56, pp. 684–700, 2016.

[5] B. Krebs, "Yahoo: One Billion More Accounts Hacked." `https://krebsonsecurity.com/2016/12/yahoo-one-billion-more-accounts-hacked/`, 2016. [Online; accessed 27-November-2018].

[6] S. Sarkar, S. Chatterjee, and S. Misra, "Assessment of the suitability of fog computing in the context of internet of things," *IEEE Trans. Cloud Computing*, vol. 6, no. 1, pp. 46–59, 2018.

[7] E. Nygren, R. K. Sitaraman, and J. Sun, "The akamai network: a platform for high-performance internet applications," *Operating Systems Review*, vol. 44, no. 3, pp. 2–19, 2010.

[8] Q. Zhang, S. Li, Z. Li, Y. Xing, Z. Yang, and Y. Dai, "CHARM: A cost-efficient multi-cloud data hosting scheme with high availability," *IEEE Trans. Cloud Computing*, vol. 3, no. 3, pp. 372–386, 2015.

[9] A. Zanella, N. Bui, A. P. Castellani, L. Vangelista, and M. Zorzi, "Internet of things for smart cities," *IEEE Internet of Things Journal*, vol. 1, no. 1, pp. 22–32, 2014.

[10] F. Bonomi, R. A. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *First edition of the MCC workshop on Mobile cloud computing (MCC@SIGCOMM 2012)*, pp. 13–16, ACM, 2012.

[11] L. M. V. González and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *Computer Communication Review*, vol. 44, no. 5, pp. 27–32, 2014.

[12] M. Chiang and T. Zhang, "Fog and iot: An overview of research opportunities," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 854–864, 2016.

[13] J. S. Plank, "A tutorial on reed-solomon coding for fault-tolerance in raid-like systems," *Softw., Pract. Exper.*, vol. 27, no. 9, pp. 995–1012, 1997.

[14] J. Matt, P. Waibel, and S. Schulte, "Cost- and latency-efficient redundant data storage in the cloud," in *10th IEEE Conference on Service-Oriented Computing and Applications (SOCA 2017)*, pp. 164–172, 2017.

[15] C. Baun, M. Kunze, J. Nimis, and S. Tai, *Cloud Computing - Web-Based Dynamic IT Services*. Springer, 2011.

[16] "Cloud computing services used by more than one out of four enterprises in the eu | eurostat." `https://ec.europa.eu/eurostat/documents/2995521/9447642/9-13122018-BP-EN.pdf/731844ac-86ad-4095-b188-e03f9f713235`, 13-December-2018. [Online; accessed 31-October-2019].

[17] P. Gupta, A. Seetharaman, and J. R. Raj, "The usage and adoption of cloud computing by small and medium businesses," *Int J. Information Management*, vol. 33, no. 5, pp. 861–874, 2013.

[18] E. A. Kosmatos, N. D. Tselikas, and A. C. Boucouvalas, "Integrating rfids and smart objects into a unified internet of things architecture," *Adv. Internet of Things*, vol. 1, no. 1, pp. 5–12, 2011.

[19] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[20] G. L. Santos, P. T. Endo, M. F. F. da Silva Lisboa Tigre, D. Ferreira, D. Sadok, J. Kelner, and T. Lynn, "Analyzing the availability and performance of an e-health system integrated with edge, fog and cloud infrastructures," *J. Cloud Computing*, vol. 7, p. 16, 2018.

[21] "Gartner identifies top 10 strategic iot technologies and trends." `https://www.gartner.com/en/newsroom/press-releases/2018-11-07-gartner-identifies-top-10-strategic-iot-technologies-and-tre` 7-November-2018. [Online; accessed 30-October-2019].

[22] "Gartner says 5.8 billion enterprise and automotive iot endpoints will be in use in 2020." `https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io`, 29-August-2019. [Online; accessed 30-October-2019].

[23] "Cisco visual networking index: Forecast and trends, 2017–2022 white paper." `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html`, 7-February-2019. [Online; accessed 30-October-2019].

[24] I. U. Din, M. Guizani, S. Hassan, B. Kim, M. K. Khan, M. Atiquzzaman, and S. H. Ahmed, "The internet of things: A review of enabled technologies and future challenges," *IEEE Access*, vol. 7, pp. 7606–7640, 2019.

[25] J. Liu, J. Li, L. Zhang, F. Dai, Y. Zhang, X. Meng, and J. Shen, "Secure intelligent traffic light control using fog computing," *Future Gener. Comput. Syst.*, vol. 78, pp. 817–824, 2018.

[26] A. A. Zaidi, B. Kulcsár, and H. Wymeersch, "Traffic-adaptive signal control and vehicle routing using a decentralized back-pressure method," in *European Control Conference (ECC 2015)*, pp. 3029–3034, IEEE, 2015.

[27] W. Chen, L. Chen, Z. Chen, and S. Tu, "A realtime dynamic traffic control system based on wireless sensor network," in *34th International Conference on Parallel Processing Workshops (ICPP 2005)*, pp. 258–264, IEEE Computer Society, 2005.

[28] J. A. Manrique, J. S. Rueda-Rueda, and J. M. T. Portocarrero, "Contrasting internet of things and wireless sensor network from a conceptual overview," in *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 252–257, IEEE, 2016.

[29] D. K. Prasad, "Adaptive traffic signal control system with cloud computing based online learning," in *8th International Conference on Information, Communications & Signal Processing (ICICS 2011)*, pp. 1–5, IEEE, 2011.

[30] K. M. A. Yousef, A. Shatnawi, and M. Latayfeh, "Intelligent traffic light scheduling technique using calendar-based history information," *Future Gener. Comput. Syst.*, vol. 91, pp. 124–135, 2019.

[31] P. Azad, N. J. Navimipour, A. M. Rahmani, and A. Sharifi, "The role of structured and unstructured data managing mechanisms in the internet of things," *Cluster Computing*, vol. 23, no. 2, pp. 1185–1198, 2020.

[32] M. Satyanarayanan, P. Bahl, R. Cáceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[33] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott, "Consolidate iot edge computing with lightweight virtualization," *IEEE Network*, vol. 32, no. 1, pp. 102–111, 2018.

[34] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos, "A comprehensive survey on fog computing: State-of-the-art and research challenges," *IEEE Communications Surveys and Tutorials*, vol. 20, no. 1, pp. 416–464, 2018.

[35] "The industrial internet consortium and openfog consortium join forces | industrial internet consortium." `https://www.iiconsortium.org/press-room/12-18-18.htm`, 18-December-2018. [Online; accessed 31-October-2019].

[36] OpenFog Consortium Architecture Working Group, "Openfog reference architecture for fog computing." `https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf`, 2017. [Online; accessed 31-October-2019].

[37] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos, "A comprehensive survey on fog computing: State-of-the-art and research challenges," *IEEE Commun. Surv. Tutorials*, vol. 20, no. 1, pp. 416–464, 2018.

[38] D. Bermbach, M. Klems, S. Tai, and M. Menzel, "Metastorage: A federated cloud storage system to manage consistency-latency tradeoffs," in *IEEE International Conference on Cloud Computing (CLOUD 2011)*, pp. 452–459, 2011.

[39] R. Rodrigues and B. Liskov, "High availability in dhts: Erasure coding vs. replication," in *Peer-to-Peer Systems IV, 4th International Workshop (IPTPS 2005)*, pp. 226–239, 2005.

[40] J. S. Plank, "Erasure codes for storage systems: A brief primer," *;login:*, vol. 38, no. 6, 2013.

[41] D. A. Patterson, G. A. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *ACM SIGMOD International Conference on Management of Data*, pp. 109–116, 1988.

[42] H. Weatherspoon and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Peer-to-Peer Systems, First International Workshop (IPTPS 2002)*, pp. 328–338, 2002.

[43] M. Schnjakin, T. Metzke, and C. Meinel, "Applying erasure codes for fault tolerance in cloud-raid," in *16th IEEE International Conference on Computational Science and Engineering (CSE 2013)*, 2013.

[44] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "*SPANStore*: cost-effective geo-replicated storage spanning multiple cloud services," in *ACM SIGOPS 24th Symposium on Operating Systems Principles (SOSP 2013)*, pp. 292–308, 2013.

[45] R. D. Pietro, M. Scarpa, M. Giacobbe, and A. Puliafito, "Secure storage as a service in multi-cloud environment," in *Ad-hoc, Mobile, and Wireless Networks - 16th International Conference on Ad Hoc Networks and Wireless (ADHOC-NOW 2017)*, pp. 328–341, 2017.

[46] R. D. Pietro, M. Scarpa, M. Giacobbe, and F. Oriti, "Wip: ARIANNA: A mobile secure storage approach in multi-cloud environment," in *2018 IEEE International Conference on Smart Computing (SMARTCOMP 2018)*, pp. 273–275, 2018.

[47] H. Gupta and U. Ramachandran, "Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access," in *12th ACM International Conference on Distributed and Event-based Systems (DEBS 2018)*, pp. 148–159, 2018.

[48] B. Confais, A. Lebre, and B. Parrein, "An object store service for a fog/edge computing infrastructure based on IPFS and a scale-out NAS," in *1st IEEE International Conference on Fog and Edge Computing, (ICFEC 2017)*, pp. 41–50, 2017.

[49] I. Althamary, C. Huang, P. Lin, S. Yang, and C. Cheng, "Popularity-based cache placement for fog networks," in *14th International Wireless Communications & Mobile Computing Conference (IWCMC 2018)*, pp. 800–804, 2018.

[50] O. Ascigil, T. K. Phan, A. G. Tasiopoulos, V. Sourlas, I. Psaras, and G. Pavlou, "On uncoordinated service placement in edge-clouds," in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2017)*, pp. 41–48, 2017.

[51] O. Skarlat, V. Karagiannis, T. Rausch, K. Bachmann, and S. Schulte, "A framework for optimization, service placement, and runtime operation in the fog," in *11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2018)*, pp. 164–173, 2018.

[52] M. I. Naas, P. R. Parvedy, J. Boukhobza, and L. Lemarchand, "ifogstor: An iot data placement strategy for fog infrastructure," in *1st IEEE International Conference on Fog and Edge Computing (ICFEC 2017)*, pp. 97–104, 2017.

[53] K. M. Konwar, N. Prakash, N. A. Lynch, and M. Médard, "A layered architecture for erasure-coded consistent distributed storage," in *ACM Symposium on Principles of Distributed Computing (PODC 2017)*, pp. 63–72, 2017.

[54] J. A. Cabrera G., D. E. Lucani, and F. H. P. Fitzek, "On network coded distributed storage: How to repair in a fog of unreliable peers," in *International Symposium on Wireless Communication Systems (ISWCS 2016)*, pp. 188–193, 2016.

[55] T. Wang, J. Zhou, X. Chen, G. Wang, A. Liu, and Y. Liu, "A three-layer privacy preserving cloud storage scheme based on computational intelligence in fog computing," *IEEE Trans. Emerging Topics in Comput. Intellig.*, vol. 2, no. 1, pp. 3–12, 2018.

[56] J. Fu, Y. Liu, H. Chao, B. K. Bhargava, and Z. Zhang, "Secure data storage and searching for industrial iot by integrating fog computing and cloud computing," *IEEE Trans. Industrial Informatics*, vol. 14, no. 10, pp. 4519–4528, 2018.

[57] I. Jacobson, I. Spence, and B. Kerr, "Use-case 2.0," *Queue*, vol. 14, no. 1, pp. 94–123, 2016.

[58] D. Leffingwell and D. Widrig, *Managing Software Requirements: A Use Case Approach*. Pearson Education, 2 ed., 2003.

[59] A. Cockburn, *Writing effective use cases*. Addison-Wesley Professional, 2000.

[60] "Iso/iec 25010:2011(en) systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models." `https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en`, 2011. [Online; accessed 01-October-2019].

[61] "Rfc4122 - a universally unique identifier (uuid) urn namespace." `https://tools.ietf.org/html/rfc4122`, 2005. [Online; accessed 01-October-2019].

[62] S. Al-Kuwari, J. H. Davenport, and R. J. Bradford, "Cryptographic hash functions: Recent design trends and security notions," *IACR Cryptology ePrint Archive*, vol. 2011, p. 565, 2011.

[63] "Martin fowler - microservices." `https://martinfowler.com/articles/microservices.html`, 2014. [Online; accessed 01-October-2019].

[64] S. Newman, *Building microservices - designing fine-grained systems, 1st Edition*. O'Reilly, 2015.

[65] T. Berners-Lee, R. T. Fielding, and L. Masinter, "Uniform resource identifier (URI): generic syntax," *RFC*, vol. 3986, pp. 1–61, 2005.

[66] C. Pautasso, O. Zimmermann, and F. Leymann, "Restful web services vs. "big" web services: making the right architectural decision," in *17th International Conference on World Wide Web (WWW 2008)*, pp. 805–814, 2008.

[67] L. Richardson and S. Ruby, *RESTful web services - web services for the real world*. O'Reilly, 2007.

[68] "Fips 197, advanced encryption standard (aes)." `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf`, 2001. [Online; accessed 01-October-2019].

[69] J. Daemen and V. Rijmen, "The block cipher rijndael," in *International Conference on Smart Card Research and Advanced Applications (CARDIS 1998)*, pp. 277–284, 1998.

[70] A. Bogdanov, D. Khovratovich, and C. Rechberger, "Biclique cryptanalysis of the full AES," *IACR Cryptology ePrint Archive*, vol. 2011, p. 449, 2011.

[71] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, pp. 120–126, Feb. 1978.

[72] "Nist sp 800-57 part 3, rev 1, recommendation for key management, part 3: Application-specific key management guidance." `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57Pt3r1.pdf`, 2015. [Online; accessed 01-October-2019].

[73] "The tls protocol." `https://www.ietf.org/rfc/rfc2246.txt`, 1999. [Online; accessed 01-October-2019].

[74] "Alexa top 1 million analysis - august 2018." `https://scotthelme.co.uk/alexa-top-1-million-analysis-august-2018/`, 24-August-2018. [Online; accessed 01-October-2019].

[75] N. Provos and D. Mazières, "A future-adaptable password scheme," in *FREENIX Track: 1999 USENIX Annual Technical Conference*, pp. 81–91, 1999.

[76] "Backblaze open sources reed-solomon erasure coding source code." `https://www.backblaze.com/blog/reed-solomon/`, 16-June-2015. [Online; accessed 02-November-2019].

[77] M. Dworkin, "Recommendation for block cipher modes of operation. methods and techniques," tech. rep., National Inst of Standards and Technology Gaithersburg MD Computer security Div, 2001.

[78] Q. Xu and J. Zhang, "pifogbed: A fog computing testbed based on raspberry pi," in *38th IEEE International Performance Computing and Communications Conference (IPCCC 2019)*, pp. 1–8, IEEE, 2019.

[79] D. Huizinga and A. Kolawa, *Automated defect prevention: best practices in software management.* John Wiley & Sons, 2007.

[80] "boxplot function | r documentation." `https://www.rdocumentation.org/packages/graphics/versions/3.6.2/topics/boxplot`. [Online; accessed 01-April-2020].

[81] X. Wang, D. Feng, X. Lai, and H. Yu, "Collisions for hash functions md4, md5, HAVAL-128 and RIPEMD," *IACR Cryptol. ePrint Arch.*, vol. 2004, p. 199, 2004.

[82] "Java virtual machine specification - chapter 1. introduction." https://docs.oracle.com/javase/specs/jvms/se14/html/jvms-1.html#jvms-1.2, 2020. [Online; accessed 01-May-2020].

[83] "Java virtual machines, java development kits and java runtime environments." http://java-virtual-machine.net/other.html, 2020. [Online; accessed 01-May-2020].