FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Secure, Cost-Efficient and Redundant Data Placement in the Cloud

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Markus Frühwirth
Matrikelnummer 00926155

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dr.-Ing. Stefan Schulte
Mitwirkung: Dipl.-Ing. Philipp Waibel

Wien, 6. November 2019

Markus Frühwirth	Stefan Schulte

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Secure, Cost-Efficient and Redundant Data Placement in the Cloud

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Markus Frühwirth
Registration Number 00926155

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dr.-Ing. Stefan Schulte
Assistance: Dipl.-Ing. Philipp Waibel

Vienna, 6th November, 2019

Markus Frühwirth

Stefan Schulte

Erklärung zur Verfassung der Arbeit

Markus Frühwirth
Loiserstraße 20, 3494 Brunn im Felde

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, 6. November 2019

Markus Frühwirth

Acknowledgements

First of all, I would like to thank my advisors Stefan Schulte and Philipp Waibel for their permanent support, invaluable feedback and discussions that have contributed significantly to the success of this work. Furthermore, I would like to thank Johannes Matt for the important exchange of information and preliminary findings during the initial phase of this work.

I also thank my friends and students for the interesting discussions and especially my girlfriend for her motivating words and constant encouragement. In particular, I would like to thank my family, who made my studies possible and supported me tirelessly and continuously during my studies.

Kurzfassung

Die Verwendung Cloud-basierter Dienste zur Datenspeicherung ist eine beliebte Alternative zu herkömmlichen Speichersystemen geworden. Die Nutzung solcher Speichersysteme kann die Datenintegrität, Datenverfügbarkeit und Datenbeständigkeit erhöhen und zugleich anfallende IT-Infrastrukturkosten senken. Immer mehr Unternehmen, Organisationen und auch Privatpersonen verwenden daher Cloud-basierte Services zur Speicherung von Daten.

Die Nutzung von Cloud-basierten Speicherlösungen bringt aber auch einige Nachteile mit sich. Die große Vielzahl der Cloud-Anbieter, die unterschiedlichen Speichertechnologien und die teilweise komplexen Preismodelle erschweren die Suche nach einem geeigneten Service-Anbieter. Eines der größten Probleme bei der Verwendung Cloud-basierter Speichersysteme ist die Abhängigkeit von einem bestimmten Anbieter, insbesondere wenn für die Datenspeicherung nur ein einziger Cloud-Anbieter verwendet wird. Solch eine Situation, bei der die Bedürfnisse der Dateneigentümer von einem bestimmten Cloud-Speicheranbieter abhängen, wird auch als Vendor Lock-In bezeichnet. Weiters wird dem Dateneigentümer durch die entfernte Datenspeicherung die physische Kontrolle über seine Daten entzogen und an den Service-Anbieter übergeben. Mögliche Sicherheitsverletzungen oder andere fehlerhafte Handlungen des Cloud-Betreibers können zu einer unerwünschten Offenlegung sensibler Daten führen.

Das Ziel dieser Arbeit ist es daher, eine Cloud-basierte Middleware zu implementieren, welche mehrere unabhängige Cloud-Speicheranbieter verwendet, um Datenobjekte sicher, authentifiziert, redundant und kostengünstig zu speichern. Um eine kosteneffiziente und dynamische Platzierung der Datenobjekte zu ermöglichen, formulieren wir einen globalen Optimierungsansatz, welcher historische Daten des Datenzugriffs berücksichtigt und vordefinierte Dienstgüteanforderungen gewährleistet. Durch die Verwendung eines effizienten Kompressionsalgorithmus wird der erforderliche Cloud-Speicherplatz verringert und somit eine Reduzierung der Gesamtkosten erzielt. Darüber hinaus wird durch die Verwendung eines authentifizierten und sicheren Verschlüsselungsalgorithmus eine hohe Sicherheit und Authentizität sichergestellt. Um eine hohe Verfügbarkeit zu erzielen und einen Vendor Lock-In auszuschließen, wird Erasure Coding als Redundanzmechanismus verwendet. Abschließend evaluieren wir den entworfenen Optimierungsansatz, indem wir ein realistisches Szenario über einen Zeitraum von sechs Monaten simulieren und die korrekte Funktionalität durch eine detaillierte Analyse der Ergebnisse belegen.

Abstract

The use of cloud-based storage services to store data is nowadays a popular alternative to traditional local storage systems. In comparison to conventional storage solutions, cloud-based storages can increase data integrity, availability and durability while lowering the overall IT infrastructure cost. Therefore, more and more businesses, organizations and even private persons started to use cloud storage systems. Especially for small and medium-sized businesses, the use of cloud storage systems can be favorable instead of maintaining their own common storage solution.

However, besides the already discussed benefits, using cloud-based storage solutions also have disadvantages. The huge amount of different cloud storage providers, storage techniques, geographical locations and pricing models makes the search for the most suitable cloud storage provider a tricky task. One of the biggest issues with using a cloud-based storage system is the reliance on the cloud storage provider itself, especially if only one is used. Such a provider can suddenly increase the price of the storage service or can go out of business. The situation, where the data owners' needs depend on a particular cloud storage provider is denoted as vendor lock-in. In addition, external storage of data removes the physical control that a data owner has over his data and forwards it to the cloud storage provider. Possible security breaches or other faulty actions of the cloud storage provider can lead to unwanted disclosure of sensitive data.

Therefore, the goal of this thesis is to provide a cloud-based middleware that uses several independent cloud storage providers to store data objects in a secure, authenticated, redundant and cost-efficient way. To find a cost-efficient placement solution, we formulate a global optimization approach that takes into account historical data access information and ensures predefined Quality-of-Service (QoS) requirements. In order to reduce the required cloud storage space and thus the overall cost, each data object will be minimized in size by an efficient compression algorithm. To ensure a high level of security and authenticity, we use an authenticated and secure encryption algorithm. To overcome the risk of vendor lock-in and to provide high data availability, we use erasure coding as redundancy mechanism. Finally, we extensively evaluate the designed optimization approach by simulating a real-world scenario over a period of six month and by proving the correct functionality through a detailed analysis of the results.

Contents

CHAPTER 1

Introduction

Nowadays, the use of cloud-based storage services to store data is a popular alternative to traditional local storage systems. In comparison to conventional storage architectures, a cloud-based solution can increase data integrity, availability and durability while lowering IT infrastructure cost [1]. More and more businesses, organizations and even private persons started to use cloud storage systems [2]. As an example, the New York Public Library, the Biodiversity Heritage Library and the United States Library of Congress began using cloud storage services in 2009 to improve both accessibility and preservation of stored data [3]. Especially for smaller and medium-sized enterprises it can be favorable to use cloud storage systems instead of maintaining their own common storage solution [2].

Because of the popularity of cloud storage systems, evermore suppliers like Amazon S3[1], Microsoft Azure[2], Google Cloud Storage[3] or RackSpace CloudFiles[4] [4] are emerging on the market. All of these vendors offer well-documented Web and API interfaces to easily store and access data while hiding the complexity of their underlying infrastructure.

1.1 Problem Definition

Besides the already discussed benefits, like the potentially higher availability and durability or the lower maintenance cost, the use of cloud-based storage systems also leads to some disadvantages.

The huge amount of different cloud storage providers, storage techniques, geographical locations and different pricing models makes the search for the most suitable cloud storage provider a non-trivial task. From the large variety of cloud storage providers,

[1] https://aws.amazon.com/s3
[2] https://azure.microsoft.com
[3] https://cloud.google.com/storage
[4] https://www.rackspace.com/cloud/files

customers have to choose the one who fits best for their own requirements. Different data guidelines or corporate policies such as data privacy and data locality make it difficult to find an appropriate cloud storage provider. As an example of data locality, a company policy must be met that requires certain data to be stored locally or at least in the same country. Or, in terms of data accessibility, frequently used data must be stored on a cloud storage provider that offers low latency and small transfer cost.

One of the biggest issues using a cloud-based storage solution is the reliance on the cloud storage provider itself, especially if only one is used. When a customer is exclusively dependent on a cloud storage provider, there are certain limitations and risks involved. This situation, in which the needs of the data owner are entirely dependent on a particular cloud storage provider, is known as vendor lock-in [5, 6]. For instance, a cloud provider can spontaneously increase the price of the storage service or can go out of business [7]. Such a plight often leads to the necessity to migrate data to another cloud storage provider, if still possible. This process can involve additional migration cost, extra time for transferring the data, further implementation and/or administration efforts. Even large cloud storage providers struggle with service outages, resulting in data being inaccessible for a period of time [1]. In the worst-case scenario, a cloud storage provider could permanently go out of business, inevitably resulting in a total data loss.

Most cloud storage vendors do not offer comprehensive data storage security guarantees. Additionally, storing data on a cloud storage leads to the loss of the physical control a data owner has. Therefore, customers have to rely on the security mechanisms and intrusion detection systems of the cloud storage provider. Furthermore, there is also the risk that the service provider may share data with other organizations or the data may be used in a way that the customer has never agreed to [8]. Even if a cloud storage provider can be considered trustworthy, administration staff or other employees with sufficient privileges can have physical access to the stored data. These so-called malicious insiders can be exposed as a well-known security problem, especially for critical information like medical records or personal data [8].

1.2 Solution Approach

A solution for the in Section 1.1 mentioned downsides is a middleware which uses several independent cloud storage providers to store data objects in a secure, authenticated, redundant and cost-efficient way. This middleware, which is based on a multi-cloud storage architecture, chooses the cheapest cloud storage provider set while respecting predefined Quality-of-Service (QoS) constraints and data access patterns.

To achieve this, the cloud-based middleware will optimize the placement of the data objects in a cost-efficient way while fulfilling several predefined QoS attributes (e.g., availability, durability and vendor lock-in factor). The middleware will also compress the data objects to reduce the overall storage size which leads to a further reduction of the storage cost. To be protected against possible security breaches by the service provider or a malicious party as mentioned in Section 1.1, the solution will encrypt each data

object by using an authenticated and strong encryption algorithm. Furthermore, the system will monitor the access information of each data object. This historical data is then used to find the most suitable cloud storage provider set. In addition, the usage of erasure coding will increase the availability and will improve the storage efficiency, which in turn reduces the total cost of the system.

1.3 Expected Results

The aim of this work is to develop a cloud-based middleware that stores data objects on multiple independent cloud storage providers in a highly available, secure, authenticated, redundant and cost-efficient way.

This work provides and evaluates an implementation of an algorithm that optimizes the placement of data objects on various independent cloud storage systems in a cost-efficient way, taking into account several predefined QoS constraints and data access patterns. In order to reduce the required cloud storage space and thus the overall cost, each data object will be minimized in terms of its size by using an efficient lossless compression algorithm. Furthermore, we will solve the security problems mentioned above by encrypting each data object with an authenticated and secure encryption algorithm. To eliminate the vendor lock-in problem, we will use erasure coding to encode the original data object and transparently distribute the resulting data object fragments across multiple cloud storage providers.

1.4 Methodological Approach

The methodological approach can be divided into three major blocks. First, in the research of the related work in the area of cloud computing, second, the extension of an already existing middleware and third, the design, implementation and evaluation of an optimization approach.

1.4.1 Survey of Related Work

Before we will start with our work, it is important to gather sufficient background information about cloud-based storages, multi-cloud services and even Peer-to-Peer (P2P) clouds. Therefore, reviewing related literature and an extensive literature research in the field of cloud computing, cloud storages, data compression, data encryption, data placement and optimization problems is essential.

1.4.2 Extension of CORA

The middleware COst-efficient data RedundAncy in the cloud (CORA) [9] is used as foundation for this work. In order to achieve our mentioned goals, we will extend the cloud-based middleware by providing the following extra features:

Data Compression Before uploading a data object to a cloud storage, it will be compressed with an efficient lossless compression algorithm to reduce the size of the required storage space. Conversely, in case of downloading a data object, the compressed data object will be decompressed to deliver the original data to the client.

Data Encryption When a data object is uploaded to a cloud storage, the data object will be encrypted with a highly secure and fast encryption algorithm and vice versa, decrypted in case of downloading a data object from the cloud storage.

Data Authentication To ensure the integrity and authenticity of a transferred data object, it will be verified by performing an authentication check.

Data Classification & Monitoring Each data object will be classified based on its object size and Multipurpose Internet Mail Extensions (MIME) type. The classification process can be imagined as a kind of grouping mechanism which measures and monitors the access information for each class of data object. This additional information will then be considered by the optimization approaches to improve the dynamic placement prediction for each data object.

1.4.3 Global Exact Optimization Algorithm

Design A global exact optimization algorithm will be designed similar to the existing solution of *Cost-optimized redundant data storage in the cloud* [10], which also used CORA [9] as foundation for their work. We will extend the design of the global exact optimization approach of [10] by taking into account the additional monitored classification information.

1.4.4 Global Heuristic Optimization Algorithm

Design A global heuristic optimization approach will be designed that takes into account the additional information monitored by the classification component.

Integration The global heuristic optimization approach will then be integrated into the middleware CORA [9].

Evaluation The evaluation of the global heuristic optimization approach will be implemented in terms of a cost analysis, based on an realistic set of cloud storage providers.

We decided to use CORA [9] because it is currently the most suitable candidate for this work. However, the system architecture of CORA [9] was basically designed to process small input objects. Furthermore, the purpose of this work is not to refactor the overall system architecture of CORA [9] itself, but rather we will focus on implementing an optimization approach for the dynamic and cost-efficient placement of the data objects.

Therefore, the only reasonably practicable approach is the implementation and evaluation of a global heuristic optimization.

We will implement and evaluate the global heuristic optimization algorithm which dynamically distributes the data objects among the most suitable cloud storage provider set. Where most suitable means to choose a subset of providers that fulfills the given QoS constraints and keeps the overall cost as low as possible. To ensure the dynamic rearrangement process between the different cloud storages, the middleware continuously monitors the access information of the data objects. In addition, each data object will be classified based on its size and MIME type to improve the placement prediction of each data object more efficiently. The global heuristic optimization approach will find an optimal or near-optimal solution within a reasonable time. The outcome of the heuristic optimization approach will be an acceptable result which is good enough to solve the dynamic placement of the data objects on several independent cloud storages.

1.5 Organization

The remainder of this thesis is structured as following:

In Chapter 2, we explain important background information which forms the theoretical base for the practical part of this thesis. We discuss some general terms and concepts of cloud computing, cloud-based storages, optimization problems, redundancy mechanism, data compression and data encryption.

Chapter 3 provides an overview of the current related work in the field of cloud-based storage solutions which uses one or multiple cloud storage providers.

In Chapter 4, we explain the underlying system model. Moreover, we introduce the concept and design of the optimization algorithms with focus on a cost-efficient dynamic placement of data objects. Then, we discuss the implementation of the designed optimization algorithms and our created components.

In Chapter 5, we evaluate the implemented global heuristic optimization approach and discuss the results. The evaluation scenario is implemented in terms of a cost analysis which is based on a realistic set of cloud storage providers.

Finally, in Chapter 6, we summarize the results of this thesis and point out possibilities for further extensions.

CHAPTER 2

Background

This chapter provides fundamental background information for this work. We will discuss the concepts of cloud-computing with its underlying service models and characteristics. Furthermore, we want to give an introduction into the important field of information security including data encryption, data authenticity and data availability. Then we want to talk about optimization problems. After that, we will explain the often used redundancy mechanism, called erasure coding. Then we will introduce several major data compression algorithms and finally, we will define and explain some QoS attributes.

2.1 Cloud Computing

Due to the continuous expansion of network infrastructures and the ongoing improvement of the underlying hardware (optical fiber cables, new transmission technologies, etc.), the data throughput of the internet is continuously increasing. So called high-speed networks and the possibility of online access from almost everywhere allow users to consume services from every place at any time over the WWW.

However, the daily growing amount of data inevitably leads to an increasing demand for storage space. More and more businesses struggle with this every increasing amount of data and the rising need for computational power [8]. As a consequence, expensive hardware and software must be regularly upgraded or renewed in order to stay always up-to-date. Additionally, this often implies increasing maintenance and operational cost. It is a fact that many businesses do not have the resources, too little premises, less money or insufficient know-how to acquire and maintain big data centres for their own needs [8].

As a result, cloud computing as an on-demand service benefits from highly growing importance over the previous years [8]. The usage of cloud computing technology facilitates organizations to manage their data and computational power needs. Using on-demand cloud computing resources allows to convert fixed cost into variable cost,

depending on the required consumption [8]. This model, often referred to as the pay-as-you-go model, benefits cloud computing compared to conventional ways with additional economic advantages [8].

2.1.1 Definition

Talking about the huge topic of cloud computing usually ends in a tricky discussion. Therefore, the U.S. National Institute of Standards and Technology (NIST) has defined a very extensive description to make the concepts of cloud computing clear.

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models. [11]

Characteristics

The NIST defines the five fundamental characteristics of cloud computing as follows [11]:

- **On-demand self-service** The cloud consumer has the possibility to change the computing capabilities based on its own needs independently from the cloud service provider.
- **Broad network access** The capabilities can be accessed over networks independent from the used client platform and device like mobile phone, tablet, laptop, workstation, etc.
- **Resource pooling** The infrastructure of the cloud provider is designed to dynamically allocate physical and virtual resources to serve multiple clients regardless of the current load. Basically, the consumer does not need to have information about the exact location of the obtained resources but he may be able to define an approximate territory or region.
- **Rapid elasticity** The cloud provider should provide scalable services which can be automatically provisioned and released based on the cloud consumers' demand.
- **Measured service** The system of the cloud provider controls and optimizes the resources appropriate to the type of service like storage service, processing service, etc. Resource usage can be monitored, controlled, and reported to ensure transparency for both, the cloud provider and cloud consumer.

Service Models

The three different service models a cloud provider can offer is described by the NIST as follows [11]:

- **Software as a Service (SaaS)** The software of the cloud provider runs as a service on its own cloud infrastructure. The service used by the consumer can be accessible from multiple different devices or via a program interface. The cloud provider has to manage and maintain the hardware, the infrastructure, different servers, storage mechanisms or some specific resources which are required to provide a high reliability of the service.
- **Platform as a Service (PaaS)** The consumer has the possibility to deploy a software solution to the infrastructure of the cloud provider. The software can be created by using programming languages, libraries, services, and tools supported by the cloud provider. Similar to SaaS, the consumer has no effort to maintain or manage the used resources of the deployed software. However, the consumer has access to some application-specific settings to control and configure the deployed software.
- **Infrastructure as a Service (IaaS)** The consumer can acquire services like processing, storage, networks or other resources to run arbitrary software on the infrastructure of the cloud provider. Unlike PaaS, the consumer has more control over the underlying infrastructure and service layers like operating systems, applications and network components.

Deployment Models

The NIST divides cloud computing services into four different deployment models [11]:

- **Private Cloud** A cloud architecture provided for an organization exclusively. It can be provisioned on or off-premises, self-managed or externally administrated.
- **Community Cloud** This deployment model is similar to the private cloud model with the difference that it can be mutually used by a specific community of different consumers and organizations.
- **Public Cloud** A cloud infrastructure which can be self-managed or externally administrated. The service can be consumed by the public and exists always on the premises of the cloud provider.
- **Hybrid Cloud** This cloud infrastructure is a composition of at least two or more different deployment models mentioned above. The different deployment models are affiliated with each other and can internally communicate over standardized interfaces that enable data and application portability.

As this thesis is mainly focused on using cloud-based storages, which is basically a special kind of service model, we will discuss this topic in more detail.

2.2 Cloud-based Storages

In general, the company that offers an online storage service to the consumer, is called the cloud storage provider. Nowadays, most of the well-known companies like Amazon (Amazon S3[1]), Microsoft (Microsoft Azure[2]), Google (Google Cloud Storage[3]), Apple (Apple iCloud [4]) or RackSpace (RackSpace CloudFiles[5]) offer their own cloud-based storage solutions. Furthermore, cloud-based storage solutions became very important in the industrial as well as in the academic research field [12]. Typical use-cases of cloud storage solutions are storing data backups, archiving files or generally using them as kind of a standard storage solution. Almost all of the cloud storage providers offer a fine-granulated file system. The consumer does not need to deal with hardware management or the complexity of different file abstraction layers [12]. As already mentioned at the end of Section 2.1, cloud-based storage solutions belong to a particular service model of cloud computing. This service model is often announced as Storage as a Service (StaaS), which can be described as a special implementation of the deployment model SaaS [13].

2.2.1 Advantages

In the following we would like to point out some essential advantages with the employment of cloud-based storages and applications [13–17].

Accessibility Storing data on cloud-based storage systems provides the possibility of access from anywhere and at any time, of course with the prerequisite of an existing internet connection. Because almost all of the cloud storage providers offer web interfaces, the data can be easily accessed by using a common web browser.

Availability Almost all cloud storage providers have implemented cost-effective redundancy mechanism which significantly minimizes potential cloud service outages. Therefore, cloud storage providers can often guarantee a Service Level Agreement (SLA) with an availability factor greater than 99.9%.

Cost One of the key benefits of cloud computing is saving Information Technology (IT) cost. Using cloud storage services instead of conventional storage systems eliminates the cost for the general storage system architecture, operational cost, software licences and fees originated by the people or organizations which are required for maintaining the local storage solution. Consumers of cloud storage

[1] https://aws.amazon.com/s3
[2] https://azure.microsoft.com
[3] https://cloud.google.com/storage
[4] https://www.apple.com/at/icloud
[5] https://www.rackspace.com/cloud/files

services have drastically less financial burden to maintain software, hardware or the general infrastructure. Experts who order hardware components, update systems or maintain software are redundant and no longer needed. Consumers pay for what they have used and release resources whenever they want. Furthermore, the cost for mechanisms or systems to provide a high level of availability and scalability can also be dropped.

Disaster recovery Saving backups to off-site cloud storages is a great advantage when local systems crash. The storage system and the included redundancy mechanism of the cloud storage provider makes the data almost always accessible. If local data gets damaged or lost, the previously stored backup data can be retrieved from the cloud storage to perform a recovery of the local system.

Scalability Using cloud based-storage services gives the advantage to only pay for the storage space that is actually needed. If businesses are growing, they can easily acquire more storage capacity keeping in mind how much they have pay to extend the storage. Conversely, if businesses are shrinking, they can release dispensable storage space at a reduced rate.

2.2.2 Drawbacks

Despite the benefits of cloud storages, questions about privacy, security, control, performance, support and vendor lock-in are raised [13–18]:

Security Sensitive business information stored on a third-party cloud service provider could potentially put a company at risk. Therefore, it is very important to choose a reliable service provider which is capable of keeping sensitive data secure. Unfortunately, security issues of cloud storage provider are a well-known problem. For instance, in 2014 Dropbox leaked confidential data due to a security glitch in their system. In 2014 too, Codespace - a major cloud data storage company, was forced to fully deactivate their services because hacker started to delete all of the customers' data.

Control If data gets migrated to a cloud storage provider, you give away the physical control of owned information. By storing data off-site and thus outside of the owner's supervision, the ability to control and customise your data storage set-up gets lost. If the required storage infrastructure is very complex, it is usually necessary to adapt the storage architecture, which cloud storage providers typically do not offer.

Vendor lock-in Once started using one particular cloud storage provider, it could be a difficult task to migrate data to another cloud storage provider. This problem, also known as vendor lock-in, occurs if the customer is fully dependent of a particular cloud storage provider. If the provider goes out of business, it may be problematic

to switch to another vendor due to the complexities of the different cloud storage infrastructures.

Bandwidth limitations Cloud-based storage systems generally depend on the speed of the available internet connection. At very slow transfer speeds, real-time data access is no longer possible or even timeout errors can occur when retrieving data.

Compliance and Restrictions by Law Depending on a organization's different regulations, it could sometimes be impossible to use a cloud-based storage solution. The same applies to normative restrictions under applicable law, which must be taken into account and assessed on a case-by-case basis. This applies in particular to healthcare, financial services and listed companies, which must be careful when using public cloud services.

Summarizing the pros and cons of using cloud-based storage solutions, in many cases the cost savings, accessibility and disaster recovery are more valuable than the associated risks. But to reduce the mentioned disadvantages, we design and implement a cloud-based middleware which stores data objects in a secure, redundant and cost-efficient way.

2.3 Optimization Problems

One of the main elements of this work are optimization problems, which generally occur in many different domains. Optimization algorithms can be called as the problem solver to find solutions which are optimal or near-optimal with respect to some goals or given constraints. Solving optimization problems in one step is usually not possible. Several processes or stages are often required to find the best fitting solution [19]. When solving optimization problems, a so-called decision alternative has to be chosen which considers all given constraints to maximize or minimize the predefined evaluation function [19]. Planning processes can be defined as the systematic, rational and theoretical process of solving optimization problems. A planning process can be divided into several steps [19]:

- **Problem recognition** In the first step, which is often described as the most difficult part, the type of problem to be solved must be identified.
- **Problem definition** After successful identification of the problem, it can be defined in detail by formulating different decision alternatives, considering additional constraints, selecting the evaluation criteria and finally defining the objectives of the planning process.
- **Problem model construction** The next step is the construction of a model, which can usually be simplified to a representative of a real-world scenario.
- **Problem model solving** The constructed model can then be solved by an optimization algorithm. An algorithm can be further defined as a procedure for passing

some tasks to find a solution with minimum or maximum evaluation value. Every algorithm starts in an initial state and should terminate in a predefined final state.

- **Solution validation** In this step, the result of the optimization algorithm must be evaluated. The validation process can be further divided into two different types of analysis, a sensitive and a retrospective approach. The first analysis method examines the dependence of the near-optimal solution on different variations of the model. The latter uses historical data to compare whether the current optimization was used in the past.

- **Solution implementation** Finally, the validated solution must be implemented into the current process.

2.3.1 Polynomial Problems

A problem which can be solved in polynomial time is called a Polynomial Problem (P-Problem). To solve a problem of size n, there is an algorithm where the number of steps within the algorithm is limited by a polynomial function of n. Usually such problems are easy to solve by using an appropriate algorithm [19].

2.3.2 Nondeterministic Polynomial Problems

A problem is called a Non deterministic Polynomial Problem (NP-Problem) if there is no known algorithm which can solve the problem in polynomial time or cannot be reduced to a P-Problem with polynomial effort. Or in other words, a NP-Problem can only be solved by guessing and verifying the solution in polynomial time, and there is no particular rule to make the guess [19].

2.3.3 Optimization Methods

The goal of every optimization problem is to find an optimal or near-optimal solution within reasonable time and effort. The solution of an optimization problem can sometimes be a compromise between solution quality and effort. Therefore, it is necessary to distinguish between two different types of optimization methods, exact and heuristic optimization methods [19].

Exact optimization methods guarantee finding an optimal solution of the input problem. Typically, exact optimization methods are very efficient for polynomial problems. The effort to solve a problem instance by using an exact optimization method grows polynomial with the problem size [19]. However, there are many optimization problems where no polynomial optimization methods are known. For example, combinatorial optimization problems like the traveling salesman problem, routing or assignment problems are NP-hard. The effort of exact optimization methods applied to such problems increases exponentially with the problem size. Therefore, exact optimization methods used for NP-hard problems are only useful for small problem instances. Larger problems or even

medium-sized problems become intractable due to the the effort that is required for solving the problem [19].

In order to conquer this problem, we can use heuristic optimization methods. Heuristic optimization methods do not guarantee that an optimal solution will be found. Usually, such optimization methods are problem-specific as they exploit the properties of the problem. In addition, they often show good performance for many NP-hard problems and problems of practical relevance [19].

2.4 Information Security

When talking about the classic model of information security, also known as Confidentiality, Integrity, Availability (CIA) triad, we can define three objectives of security: confidentiality, integrity, and availability. Each objective addresses a different aspect for providing protection of information [20]:

Confidentiality The ability to protect information from disclosure to unauthorized parties. Only parties who are authorized and allowed to access the data can read the sensitive information.

Integrity The ability to ensure the authenticity of the information. This includes guaranteeing that the information is not altered and the source of the information is genuine.

Availability The ability to make the information available to authorized users at any time.

Ensuring data availability is nowadays one of the most sophisticated task when talking about the CIA triad [20]. Denying access to resources via Denial of Service (DoS) or Distributed Denial of Service (DoS) attacks has become a very common problem. There are currently some mechanisms which are resistant to such attacks, but they are often too expensive. A simpler option is to regularly back up data or provide redundancy mechanisms. For example, regarding to the latter, if someone floods a companies' primary data center with requests via a DoS attack, it may be possible to switch to a second, redundant data center to reduce downtime.

2.4.1 Encryption

$\longrightarrow$ *Ensuring the* ***C*** *of the CIA triad*

Besides providing high availability and reliability, most of the cloud storage providers only ensure a certain base-level of security. No matter how trustworthy a cloud storage provider might be, there is always the risk that data will be leaked to public, resold to other companies or shared in a way that the data owner never intended. Therefore,

sensitive data should be encrypted on the client side before an upload to a cloud storage takes place.

Nowadays, the process of digitisation has reached most of the major industries. Bank accounts, personal information, credit cards, trade secrets, online documents, private information, health information or in general, a major part of the digital information has to be kept secret. The protection of such data is therefore one of the largest and most important areas of information security. Encryption algorithms, which play a decisive role in information systems, are an important solution for ensuring confidentiality. In general, encryption can be defined as the process of scrambling sensitive data in a way that only the intended recipient can read it. Encryption algorithms can basically be classified into two major groups [21]:

Symmetric key Encryption The conventional encryption method, which uses the same key for the encryption and decryption process.

Asymmetric key Encryption Also known as public-key encryption, which uses two different keys for the encryption and decryption process, especially a public key and a private key.

In general, asymmetric key encryption techniques are much slower from scratch then symmetric algorithms. If we compare both techniques by applying the same secret key, the symmetric algorithms are much more efficient to achieve the same level of security [21]. Therefore, we will discuss only some of the major symmetric encryption techniques relevant to this work [21]:

Data Encryption Standard (DES) is a block cipher designed by IBM in 1977 to encrypt and decrypt 64 bit blocks with a 64 bit input key, whereas only 56 bits are used. The algorithm performs 16 iterations to convert plain text into cipher text. Conversely, the same steps are used for the decryption process. DES is considered as an insecure block cipher because there exist a huge amount of exploits. DES is very vulnerable to brute force attacks as only 2^{56} imaginations are required. Besides the known weakness of DES, the algorithm is widely used by financial services and other industries.

Triple Data Encryption Standard (TDES) or Triple Data Encryption Algorithm (TDEA) is an extension of DES, which was mainly developed to protect against brute force attacks. TDES extends the key size of DES by using three keys of the length of 168 bits ($3 \cdot 56$ bits). TDES applies the algorithm of DES three times in succession. In more detail, TDES uses three 64 bit keys k_1, k_2 and k_3 in Encrypt-Decrypt-Encrypt (EDE) mode. This means that the plain text is encrypted with k_1, then decrypted with k_2 and finally encrypted with k_3. TDES uses a block size of 64 bits and performs 48 ($3 \cdot 16$) processing rounds. TDES is conceptualized to eliminate the weakness of DES by accepting a much more time-consuming encryption process.

Advanced Encryption Standard (AES) / Rijndael has been announced as the encryption standard recommended by NIST and replaced DES in 2001. AES is generally based on a substitution and permutation network. It has a fixed block size of 128 bits and three possible key sizes of 128 (AES-128), 192 (AES-192) and 256 bits (AES-256). Dependent on the key size, AES performs 10 rounds for AES-128, 12 rounds for AES-192 and 14 rounds for AES-256 to deliver the final cipher text. AES is a very flexible encryption standard and has an outstanding high throughput.

Blowfish Blowfish is a symmetric block cipher algorithm, accepted as a fast and strong encryption algorithm. Blowfish is a fixed 64 bit block cipher and takes an input key length from $32 - 448$ bits. Blowfish processes a total of 16 rounds of data encryption. Blowfish is patent-free, license-free, and available to everyone for free.

Twofish Twofish can be treated as encryption algorithm derived from Blowfish. Twofish is a 128 bit block cipher with possible key sizes of 128, 192 and 256 bits and the same amount of processing rounds as Blowfish. It uses precomputed key-dependent substitution-boxes, so called S-Boxes, to obfuscate the relationship between the key and the ciphertext by using permutation and a complex key schedule. Twofish is very fast, flexible and is designed to made it easy to extend keys and processing rounds up to 124.

Beside the various existing encryption algorithms, the selection of the key is a very important task, since the security of the encryption algorithms depends directly on it. The strength of the encryption algorithm depends mainly on the secrecy and length of the key. Such a key can be represented by a numeric text, an alphanumeric text or a special symbol. However, the encryption and decryption process often requires enormous computing power, processing time and storage space.

2.4.2 Authentication

$\longrightarrow$ *Ensuring the* ***I*** *of the CIA triad*

When talking about the authenticity of information, we have to consider two aspects: data integrity and data origin authentication. Whereby, data integrity means that the information has not been modified in an unauthorized fashion since its creation, transmission or retention by an authorized origin. On the other hand, data origin authentication is the method of ensuring the source of the information [22].

Hash Functions

A commonly used method to preserve data integrity makes usage of one of the most important mathematical algorithm in cryptography, the so-called hash function. Because of the special one-way characteristic, hash functions are essential in the field of software security. When applying a hash function to data of arbitrary size, the result is a bit

string of a fixed size which is infeasible to invert. With reference to this work, if a data object is uploaded to a cloud storage provider, you can apply a hash function on this data object. The result, the hash, can then be stored securely in a database. Later on, if the same data object is downloaded from the cloud storage provider, the prior used hash function can be applied on the received data object. If both hash values are matching, you can be sure that the data was not modified during the transmission. A positive side effect of such an authenticity check is that this process automatically includes an error detection. If the received data became corrupted during the data transport, the compared hashes cannot be equal [22].

Message Authentication Code

A common algorithm to ensure the authenticity of information is a Message Authentication Code (MAC). Before data is sent to a receiver, the sender computes a so-called authentication tag which is usually appended to the transmitted data. In general, such an authentication tag is computed by a function that takes into account the data itself and a common secret key. On the recipient side, the equivalent function is reapplied to compute a new authentication tag. If both match, the data can be considered valid. Since only the sender and recipient know the shared secret key, the origin can also be assumed to be trustworthy [22].

There are several different ways how to generate a MAC. It can be computed by using hash functions (e.g., Hash-based Message Authentication Code (HMAC)), universal hash functions (e.g., Universal-hashed Message Authentication Code (UMAC)) or block ciphers (e.g., One-key Message Authentication Code (OMAC), Cipher Block Chaining Message Authentication Code (CBC-MAC), Parallelizable Message Authentication Code (PMAC)) [22].

2.4.3 Authenticated Encryption

$\longrightarrow$ *Ensuring the* ***C*** *and* ***I*** *of the CIA triad*

Authenticated Encryption (AE) ciphers provide confidentiality, integrity, and authenticity and are excellent in respect to performance and power efficiency [23]. Especially, Authenticated Encryption with Associated Data (AEAD) ciphers, which are context sensitive, add the possibility to check the integrity and authenticity of Associated Data (AD). Therefore, the injection of a valid ciphertext into another context will always be detected by the algorithm.

Galois/Counter Mode

Galois/Counter Mode (GCM) is special mode of operation for symmetric key block ciphers. GCM is designed for throughput rates in high-speed communication channels with moderate hardware resources and is also ideally suited for software implementations.

GCM is designed for block ciphers with a block size of 128 bits and thus ideal for AES. Moreover, GCM can be applied in an authentication-only mode, which in this case simply acts as an incremental MAC, called Galois Message Authentication Code (GMAC). In simple terms, a GMAC uses a universal hash function that is defined over a binary Galois field [24].

AE algorithms which use GCM or GMAC provide a very strong authentication assurance for the ciphertext as well as the not encrypted AD. Therefore, GCM is highly recommended when using AEAD algorithms [23].

AES-GCM

The majority of the communication over the internet uses the security protocol Secure Sockets Layer (SSL), or the newer and more secure one Transport Layer Security (TLS). AES-GCM is an AEAD cipher that is used in TLS 1.2 and provides both data integrity and data origin authentication [23].

2.5 Erasure Coding

$\longrightarrow$ *Ensuring the* ***A*** *of the CIA triad*

Most of the current cloud storage providers have applied erasure codes to their storage systems to ensure high availability and reliability while reducing storage overhead [25, 26].

In a conventional and easy to setup full replication storage system, a data object is replicated and put to different storage nodes. The amount of storage nodes used can be assumed as the multiplier of increasing reliability. For example, if you want to achieve a four times higher reliability of your current storage system, you have to use four additional storage nodes which hold the replicated data. Each replica occupies the same storage capacity as the original data object. The system fails, or in a storage system you can say, the data is lost, if all the storage nodes crash simultaneously. However, this redundancy mechanism needs too much storage overhead to achieve high durability [27].

Instead of a simple replication mechanism, erasure coding can be used as an alternative way to provide redundancy. Erasure coding provides redundancy while reducing the storage overhead of strict replication with the advantage of the same or a higher level of data reliability [25, 27]. This can be achieved by disassembling a data object into m data fragments of the same size. This m data fragments are then encoded into n data fragments whereas any subset of m, with $m < n$ data fragments can be used to reconstruct the original data object [25]. Erasure coding is often defined by the double-tuple (m, n) [9]. The benefit of such an erasure coded data object is that it can tolerate the loss of $k = n - m$ data object fragments while still being able to reconstruct the original data object [9].

Further we can call $r = \frac{m}{n} < 1$ as the *rate* of encoding. The inverted *rate* $\frac{1}{r}$ can be used as enhancement factor for the storage cost. For example, a system with an applied

Table 2.1: Comparison: RAID Levels with Erasure Coding

RAID Level	Erasure Code
1	$EC(1, 2)$
4	$EC(1, 4)$
5	$EC(1, 4)$

erasure code $(1, 4)$, often written as $EC(1, 4)$, has an encoding *rate* of $r = \frac{1}{4}$ which increases the storage cost by the factor of $\frac{1}{r} = 4$ [27].

Erasure coding can also be called as a superset of fully replication and Redundant Array of Independent Disks (RAID) systems [27]. For instance, in the example above we introduce an erasure code of $EC(1, 4)$ which describes a fully replication system with $n = 4$ replicated storages. The different RAID levels can also be defined as shown in Table 2.1.

The biggest advantage of using erasure coding against replication (RAID) is the smaller additional amount of storage space needed to achieve the same level of redundancy [9, 28].

2.6 Data Compression

In general, data compression is the method of lessening the size of the data without a significant loss of information. If we talk about data compression, we have to distinguish between the following two categories, lossy and lossless compression [29]:

Lossy compression permanently reduces the size of data by removing certain information, preferably redundant information. It is important to know that this process is irreversible, which means that the original data cannot be recovered when the data gets uncompressed. Lossy compression algorithms reduce the size of data significantly while raising quality loss. Therefore, such algorithms are mainly used where the loss of certain information is acceptable, especially for compressing images and videos [29].

Lossless compression temporary reduces the size of data by removing certain information, preferably redundant information. In contrast to lossy compression, lossless compression algorithms restore the original data after the decompressing process. Such algorithms are used where it is important to recover the original data [29].

Data compression plays an important role in the field of Web engineering and data transmission. Web developers, User Experience (UX) designers, software architects and network experts are particularly interested in accelerating data transfer from the server to the clients. An effective way to achieve this is to reduce the size of transferred data with fast and efficient compression algorithms. Most often, the process of data compression is memory- and CPU-intensive. Thus it is important to select the correct algorithm in

terms of parameters such as compression density, processing time and system resource consumption [29].

Many different compression algorithms are developed, each with different characteristics. Some of them focus on keeping the compression time as short as possible while producing an acceptable compression rate. Conversely, some of them specialize in high compression rates, with the disadvantage of long processing times. In this work it is important to always restore the original file. Therefore, we are limited to using only lossless compression algorithms.

The most popular and widely-used lossless compression format over many years is GNU zip (gzip) [30]. The gzip compression format is an extension of the well-known DEFLATE algorithm, which is a combination of both, the LZ77 (Lempel–Ziv, 1977) dictionary-based algorithm and Huffman coding. To facilitate the integration of gzip into applications for software developers, Gailly and Adler have developed the zlib library, which is widely used and very effective [29]. The fast processing times coupled with a good compression ratio, especially for text files [30], are the key factors for the popularity of gzip.

CHAPTER 3

Related Work

This section gives an introduction to the most important related work of this thesis. First of all, we will discuss CORA [9], the core system we have used and extended for this work. Another fundamental work for this thesis is CORA 2.0 [10], which can be announced as the successor of CORA [9]. Subsequently, we talk about some multi-cloud platforms that use different optimization approaches for (cost-efficient) redundant data storage mechanisms taking into account predefined service constraints. Then we refer to some other solutions and basic approaches which are important for this work. At the end of this section, we will discuss the existing approaches in relation to our work.

3.1 Fundamental Work

3.1.1 CORA [9]

As has already been mentioned in Section 1.3, CORA [9] is used as foundation for this thesis. CORA [9] is a cloud-based middleware which uses multiple cloud storage providers to guarantee a cost-effective and redundant storage of data objects. The middleware records the access information of the data objects and finally selects the most suitable provider set while taking into account several predefined attributes and QoS constraints. The predefined data attributes and QoS constraints are always fulfilled by the middleware while guaranteeing a cost-efficient data placement on the cloud storages. Availability, durability and the level of vendor lock-in are just a few examples that can be set to a predefined value by the client. In addition, erasure coding is used to split the data objects into several data fragments to store them on multiple different cloud storages in a RAID-like fashion. The cost-reducing optimization approach is implemented based on the Mixed Integer Linear Programming (MILP) technique. MILP can solve problems where only some of the variables need to be integers, while the others can be non-integers. CORA [9] also considers different storage types and complex pricing models (e.g., Block Rate Pricing (BRP)) to optimize the data placement.

3.1.2 CORA 2.0 [10]

CORA 2.0 [10] can be announced as the successor of CORA [9] and likewise as the predecessor of this thesis. CORA 2.0 [10] also uses CORA [9] as core middleware for their work. CORA 2.0 [10] extends CORA [9] by adding three optimization approaches. They extend the existing local optimization approach to global scope while not modifying the core implementation of CORA [9]. One approach optimizes the data placement of the data object fragments using the MILP technique and considers the history information of each data object which is monitored by the system. Another optimization approach uses a heuristic function, the well-known knapsack problem instead of MILP. A further optimization is done by additionally considering real-time measured latencies which occur during a read or write request to the cloud storages.

Both CORA [9] and CORA 2.0 [10] neither apply compression algorithms on the data objects to reduce the required cloud storage, nor encrypt the data to ensure high security of the information. Furthermore, they do not provide classification of data objects to predict a cost-efficient placement solution on the cloud storages.

3.2 Further Work

3.2.1 Scalia [31]

Scalia [31] reduces cost by providing redundant data storage on multiple clouds. In order to achieve this, the system periodically checks the access information of the data that is stored on the cloud providers. Based on this information history, Scalia [31] checks if a rearrangement of the data to adequate providers is more cost-efficient. If so, the system performs the optimization by taking into account several predefined user constraints. The system uses erasure coding for splitting data objects into several data fragments to store them independently on different cloud storages. The distribution of the data fragments on the cloud storages is based on a heuristic function, the well-known knapsack problem, which is also in the scope of this thesis. Scalia [31] uses data classification based on the MIME type and/or file size and collects statistics regarding the resources used to predict the lifetime of a new object at the time of insertion. In addition, Scalia [31] improves the performance of read requests by providing a caching layer. Simultaneously, this layer reduces cost, as in case of a read request the system must not necessarily initiate a read request to the cloud storage provider. Therefore, if the requested data can be found locally by the caching layer, Scalia [31] can immediately return the data object without the need of an expensive Application Programming Interface (API) call. This reduces cost, time and computational power of the whole system.

However, Scalia [31] does not use a long-term storage solution to exploit the lower storage price and uses a simplified pricing model instead of BRP, which makes the cost-reducing optimization approach across different cloud storages error-prone. Scalia [31] does not use data compression to reduce the required amount of cloud storage space, and does not use encryption algorithms to provide security and authenticity.

3.2.2 CHARM [25]

CHARM [25] (Cost-efficient Data Hosting Model With High Availability In Heterogenous Multi-Cloud) is also a multi-cloud platform which distributes data over several predefined cloud storage providers in a cost-efficient way in order to guarantee flexible availability and durability, but mainly to avoid vendor lock-in. CHARM [25] combines two redundancy mechanisms, full-replication and erasure coding, into a uniform model to meet the required availability in the presence of different data access patterns. The system also monitors the access history of each data object to choose the most suitable redundancy mechanism for a particular data object. Or in other words, CHARM [25] automatically selects the cheapest redundancy mechanism for a particular data object. The system uses a heuristic function for the data placement on cloud storages. The used function is called the Kernighan-Lin algorithm, which is a heuristic algorithm for solving the graph partitioning problem (e.g., the travelling salesman problem).

CHARM [25] uses a simplified pricing model that makes no difference between standard and long-term storage solutions and does not consider BRP models. In addition, CHARM [25] has not implemented any data compression mechanisms or encryption algorithms to provide a secure and more cost-efficient storage solution. Moreover, CHARM [25] provides no classification of data objects to predict a cost-efficient storage solution.

3.2.3 SPANStore [32]

SPANStore [32] (Storage Provider Aggregating Networked Store) is a multi-cloud storage system that automates the process of reducing cost and latency while satisfying consistency, flexibility and fault tolerance requirements. SPANStore [32] is designed to efficiently determine where a data object should be replicated and how this replication should be performed by taking into account replication policies based on workload properties and by minimizing the use of computer resources. Generally, SPANStore [32] minimizes cost by exploiting pricing discrepancies between cloud storage providers.

However, SPANStore [32] does not make use of erasure coding to redundantly store data objects on multiple cloud storage providers. Furthermore, SPANStore [32] does not use long-term storages to minimize cost and does not use BRP models. The system has neither security nor compression mechanisms implemented. Finally, SPANStore [32] does not have an integrated classification mechanism to predict a cost-efficient storage distribution of data objects over several cloud storages.

3.2.4 DepSky [33]

Bessani et al. [33] propose a virtual multi-cloud storage system architecture called DepSky (Dependable and Secure Storage in a Cloud-of-Clouds). The system is generally focused on guaranteeing a high level of confidentiality, integrity, authenticity and security. DepSky [33] uses erasure coding as redundancy mechanism and distributes the data

object fragments among several cloud storages in order to conquer the vendor lock-in problem. DepSky [33] consists of two separate algorithms, DEPSKY-A and DEPSKY-CA, whereas the latter encrypts and afterwards encodes each data object before uploading the data object fragments to the different cloud storages. The multi-cloud architecture of DepSky [33] uses a combination of Byzantine fault tolerance algorithms, secret sharing mechanism and erasure coding to provide a high level of availability and confidentiality.

But, DepSky [33] does not aim on minimizing the overall cost and is limited to only four clouds, where each cloud uses its own specific interface. Furthermore, DepSky [33] does not use BRP models, but rather makes use of a simplified linear pricing model respectively to the cloud storage space as calculation base of the total storage cost. In addition, DEPSKY-A does not use encryption algorithms to ensure high security of information and does not take into account the cost advantages between standard and long-term storages. DepSky [33] does not use compression techniques to reduce the file size of data objects and does not use a method to classify data objects to predict a cost-efficient placement solution.

3.2.5 RACS [6]

The main focus of RACS [6] (Redundant Array of Cloud Storage) relies on implementing RAID, especially RAID5, at cloud level to provide high availability and efficient data replication across multiple cloud storages. In general, the main objective of RACS [6] is to avoid vendor lock-in and its associated risks. The system uses erasure coding to accomplish a reliable and efficient RAID-like data storage solution. To achieve this, RACS [6] acts like a proxy sharing data fragments across multiple cloud storage providers in a transparent and economical way. RACS [6] further reduces storage cost by dynamically selecting cloud storages based on the cloud storage provider's different cost models.

RACS [6] does not use long-term storages, which has a big impact on the total storage cost. RACS [6] does not solve security problems caused by the use of cloud storages and does not provide an authentication mechanism that ensures confidentiality and integrity. In addition, the system does not use a compression algorithm to efficiently reduce the cloud storage space required on the cloud storages and does not use a classification method to achieve a more sophisticated placement solution of data objects.

3.2.6 Cloud-RAID [8]

Schnjakin et al. [8] introduce a multi-cloud system that provides privacy, authenticity and redundancy by splitting data objects into multiple same sized fragments and distributing them to multiple cloud service providers. The system uses erasure coding to provide a RAID-like and fail-safe storage solution and is mainly designed to compensate the performance of slow cloud storage provider. If slow provider throughput is detected, the system optimizes the rearrangement of the data object fragments by moving more

fragments to a faster service provider. Cloud-RAID [8] makes use of AES encryption and Secure Hash Algorithm (SHA) to ensure data security and data integrity.

However, Cloud-RAID [8] does not use an optimization approach to store data in a cost-efficient way and does not exploit the cost-benefits of long-term storages to minimize overall cost. The system has not implemented a compression algorithm. Furthermore, Cloud-RAID [8] does not use a classification mechanism to predict a cost-efficient placement solution for data objects.

3.2.7 HAIL [34]

HAIL [34] (High Availability and Integrity Layer) manages data redundancy across multiple cloud providers. The system uses erasure coding for a redundant data storage on multiple cloud storages. HAIL [34] is basically designed as a distributed cryptographic system which proves whether a stored file is not erroneous and accessible to the client. Thus, the system provides a software layer designed to achieve high availability and integrity of the stored data. HAIL [34] verifies the availability and integrity of a file by checking the corresponding file parts at multiple distributed servers. If the system detects a corruption on a particular server, it forces the remaining servers to recover the corrupted file.

HAIL [34] has not implemented an algorithm to store data on multiple cloud storages in a cost-efficient way and does not use long-term storages. Furthermore, the system does not provide any security mechanism or compression methods and requires a cloud server to run the application. Moreover, HAIL [34] does not have an integrated classification mechanism to predict a cost-efficient storage distribution of data objects.

3.3 Discussion

An overview of the feature-comparison of the presented related work is shown in Table 3.1. In summary, most of the mentioned systems are aiming at a cost-efficient placement solution of data objects, and almost all of them use erasure coding to provide redundancy. Only CORA [9] and CORA 2.0 [10] consider BRP models in their cost calculation to find the most suitable storage solution.

With the exception of CORA [9] and CORA 2.0 [10], none of the mentioned systems offer a cloud-based storage solution that utilizes the cost efficiency of long-term storages by taking into account previously-monitored access information of data objects. Besides, only DepSky [33] (e.g., DepSky-CA) and Cloud-RAID [8] encrypt data objects to ensure high security and perform authentication checks to provide authenticity. None of the mentioned approaches use compression algorithms to reduce the amount of cloud storage space required.

Table 3.1: Related Work: Feature Comparison

✓ Feature is provided
- Feature is not provided

Feature	**SCORA**	CORA [9]	CORA 2.0 [10]	Scalia [31]	CHARM [25]	SPANStore [32]	DepSky [33] (A)	DepSky [33] (CA)	RACS [6]	Cloud-RAID [8]	HAIL [34]
Erasure Coding	✓	✓	✓	✓	✓	-	✓	✓	✓	✓	✓
Cost Optimization	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	-
Block Rate Pricing (BRP)	✓	✓	✓	-	-	-	-	-	-	-	-
Long-Term Storage	✓	✓	✓	-	-	-	-	-	-	-	-
Data Access Pattern	✓	✓	✓	✓	✓	✓	-	-	-	-	-
Data Classification	✓	-	-	✓	-	-	-	-	-	-	-
Data Encryption	✓	-	-	-	-	-	-	✓	-	✓	✓
Data Compression	✓	-	-	-	-	-	-	-	-	-	-

3.4 Challenges

This work will provide a heuristic optimization algorithm that stores data objects in a redundant and cost-efficient way. The algorithm will consider real price models (e.g., BRP models) of several cloud storage providers and will differentiate between standard and long-term storages to take advantage of the cost-effective characteristics of both. Furthermore, the middleware will ensure that predefined data access patterns are always fulfilled. Erasure coding will be used to provide a redundant storage solution. In addition, each data object will be compressed by an efficient lossless compression algorithm to reduce the required cloud storage space. The middleware will encrypt each data object by using a secure encryption algorithm to achieve a high level of security. Finally, our system will provide data integrity and authenticity by using a MAC.

CHAPTER 4

Design

In this chapter, we will first introduce the system architecture of this work and the underlying cloud-based storage middleware CORA with its basic functions. Later on, we will present our extended version of CORA with a greater detail to our newly designed components. Then, we will describe the system model with their appropriate terms and definitions. Finally, we will define the optimization approaches one by another.

4.1 System Architecture

In this section, we will explicitly describe the underlying system architecture. First, we will give a simplified overview of the used cloud-based middleware CORA. Then, we will explain the functionality of each major component more precisely. Further on, we will present the system architecture of Secure, COst-efficient data RedundAncy in the cloud (SCORA), our extended version of the middleware CORA.

4.1.1 Simplistic Model

Figure 4.1 illustrates a very simplified version of the overall system architecture. At the very bottom of the figure is the client user, who wants to initiate CRUD requests to the data object fragments stored on multiple cloud storage providers. Since we do not want to be dependent on a single cloud storage provider, or rather we want to avoid a vendor lock-in, we use several independent cloud storage providers. This so called multi-cloud storage architecture is shown on top in Figure 4.1. In the center of the image, between the multi-cloud architecture and the client user, you can see the cloud storage middleware. This middleware includes all the logic which is necessary to interact with the APIs of the various cloud storage providers. It handles the incoming requests from the user, performs data processing and finally, forwards the corresponding commands to the cloud storages.

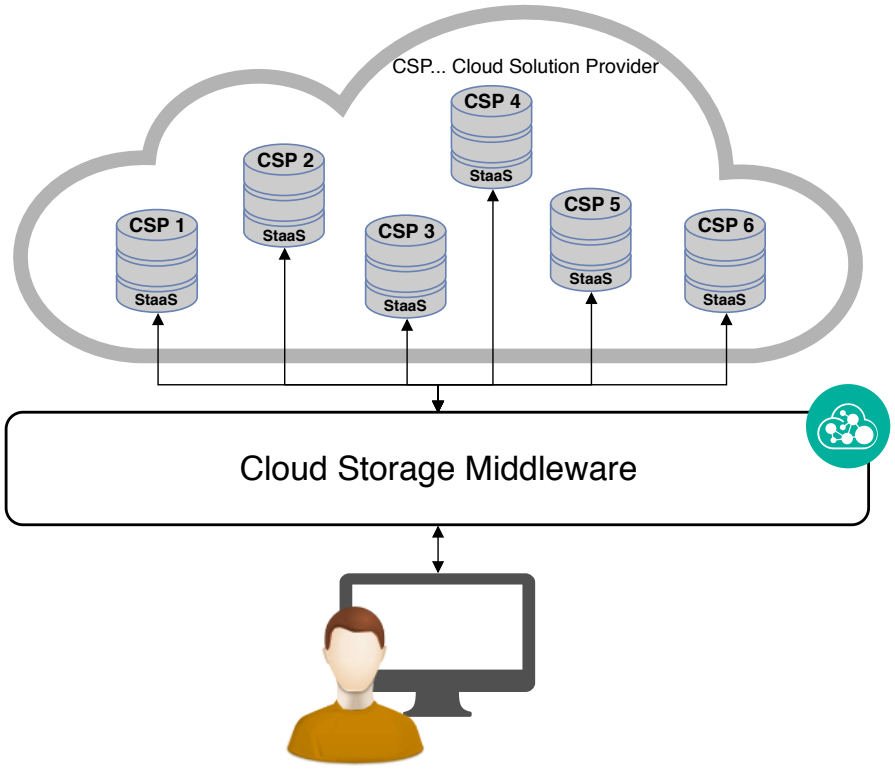


Figure 4.1: Simplified System Architecture

4.1.2 CORA

As we already have mentioned in Section 3.1.1, we use the middleware CORA as foundation for this work. We decided to use CORA because many functions such as the compatibility of multiple cloud storage APIs, erasure coding to split data objects into commensurate fragments, data access monitoring, and the possibility to integrate different placement optimization approaches are still provided. To provide a more detailed overview of the system architecture of CORA, which can be seen in Figure 4.2, we will describe the most important functional components.

4.1.2.1 API

The API component of CORA consumes the requests from the client user and provides the standard CRUD endpoints for performing read, write, update and delete operations on data objects. The received requests are processed by the API component and afterwards forwarded to the Data Manager.

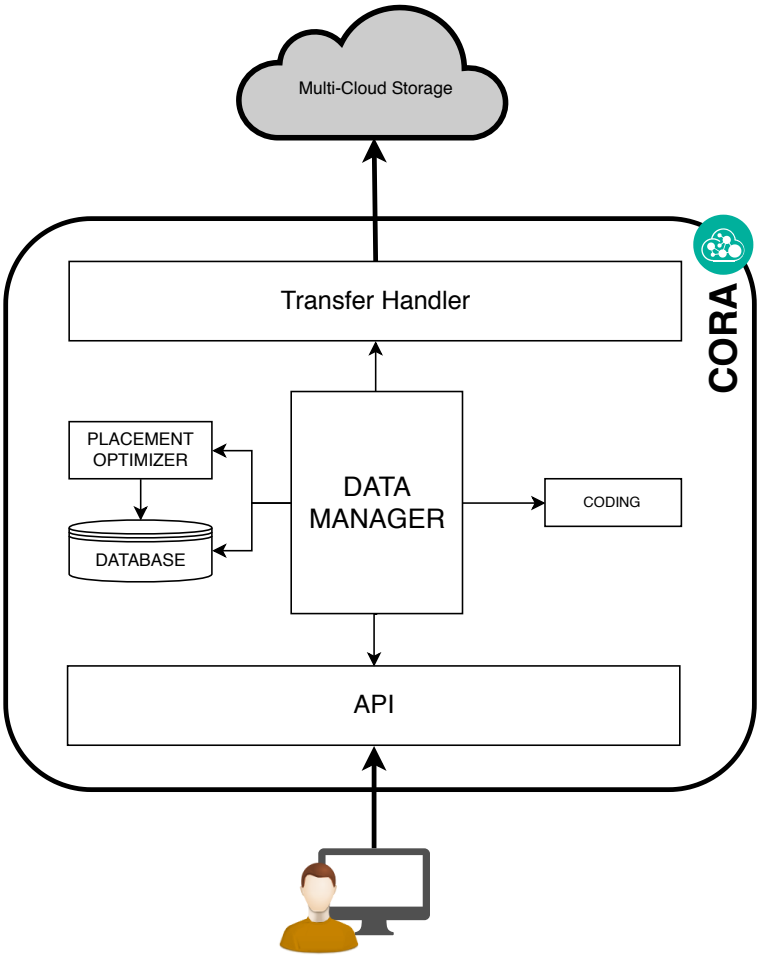


Figure 4.2: System Architecture of CORA

4.1.2.2 Data Manager

The central component of CORA is the Data Manager, which gets the instructions from the API component, coordinates the communication between the other components and handles the respective CRUD commands. The Data Manager is also responsible for monitoring all the actions performed on the data objects and stores this access information into the Database of CORA.

4.1.2.3 Coding

As has already been announced in Section 2.5, CORA uses erasure coding to achieve a certain degree of redundancy across multiple cloud storages. This RAID-like fashion to store data objects requires several coding operations. The Coding component of CORA performs the necessary operations and applies erasure coding, defined as $EC(m, n)$ with

the predefined parameters m and n, on the data objects.

For example, if a client initiates an upload request of a data object o via the corresponding API endpoint of CORA, the Coding component splits and encodes the data object o into n commensurate data fragments $o_1, o_2, \ldots, o_n$, which are then distributed to different cloud storages. Reversely, if the client wants to read a data object o, the Coding component reads the data fragments $o_1, o_2, \ldots, o_n$ from the cloud storages and joins them to a single data object o. Because CORA optimizes the cost from scratch, the middleware does not need to read all n data object fragments from the cloud storages. More precisely, it only needs to read m fragments that are enough to restore the whole data object o. Furthermore, the Coding component detects corrupt or empty (e.g., if a cloud storage provider is out of service) data object fragments and recreates the original, if necessary.

4.1.2.4 Optimizer

As the name already suggests, the Optimizer component is responsible for the cost optimization processes of CORA, or more precisely, this component organizes the placement of the various data object fragments to reduce the overall cost and besides considers predefined Service Level Objectives (SLOs). By default, the optimization process is invoked if any of the following events occur:

- A read, write or update request on a data object
- The set of available cloud storages is changed
- A predefined recurring optimization event is called

The optimization algorithm of CORA finds the least expensive placement solution for a data object. To be more exact, it selects the cheapest set of independent cloud storages to store several related data object fragments. During the optimization process, the applied SLOs such as level of availability, durability or vendor lock-in are guaranteed. The optimization algorithm of CORA uses historical access information to predict the future usage of a data object. The provided optimization algorithm of CORA is locally oriented, which means that an optimal placement solution can only be found in the scope of each individual data object. Moreover, if the Optimizer recognizes that a data object is rarely or not used in a certain period, it will immediately initiate a migration process for transferring this data object to a long-term storage.

4.1.2.5 Database

The Database component of CORA holds the access and storage information of each data object. As has already been mentioned in Section 4.1.2.4, this historical access information is important for the Optimizer. Since the Database component of CORA is designed as an In-Memory Database (IMDB), long system runtimes or the processing of large amounts of data objects require enormous memory space which in turn slows down the overall system.

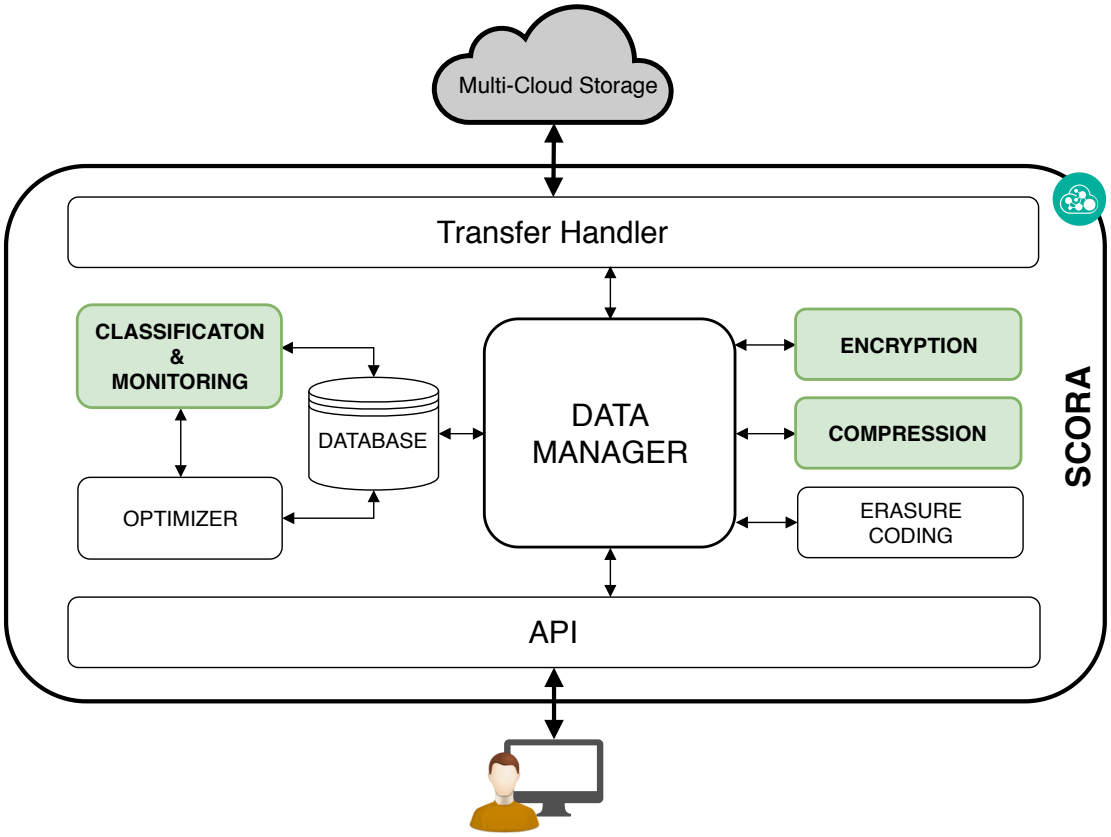


Figure 4.3: Extended System Architecture of CORA

4.1.2.6 Transfer Handler

This component receives the commands from the Data Manager, as mentioned in Section 4.1.2.2, and forwards them to the cloud storages. Basically, the Transfer Handler can be treated as an interface component between CORA and the cloud storages.

4.1.3 Extension of CORA

We extend the middleware CORA by adding three functional blocks. One of them is the Encryption component, which is responsible for encrypting and decrypting the data objects. The second block is the Compression component, which compresses and decompresses data objects to reduce the file size and thus the required cloud storage space. The last block is responsible for the data object classification. Each data object gets classified based on its data object size and/or MIME type. Our extended system architecture, called SCORA, is shown in Figure 4.3. Each component is self-contained and independently designed.

4.1.3.1 Compression

If a user initiates an upload request of a data object, the Compression component reduces the content length by applying an efficient and fast compression algorithm. However, depending on how the binary data of the source data object is assembled and which compression algorithm is used, the compressed data object may be larger than the uncompressed one. Therefore, the Compression component compares both, the original and the compressed size of the data object and chooses the smallest outcome. Thereby, the Compression component ensures that the size of the compressed data object is always smaller or equal than the original data object. Conversely, in case of a download, the Compression component decompresses the compressed data object to deliver the original file to the user. Furthermore, the Compression component is designed in such a way that it offers the possibility of integrating different compression algorithms for compressing the data objects.

Because of the mentioned advantages in Section 2.6 and the results of [29] and [30], we decided to integrate gzip as base compression algorithm for this work.

4.1.3.2 Authenticated Encryption

The Encryption component encrypts each data object before an upload to the cloud storages takes place. This ensures that each uploaded data object fragment is always stored encrypted on the cloud storages. For every upload or update request of a data object, the component generates a new random secret key and finally stores this value into the Database. Therefore, a very high security level can be reached. Reversely, if the user downloads a data object, the corresponding secret key gets fetched from the IMDB and then decrypted by the component to deliver the original data object.

Besides of using a strong encryption algorithm, we further want to ensure data authenticity and integrity. To achieve this, we make use of a special kind of encryption algorithms, which are called AEAD. After a deep literature research in the field of encryption algorithms [21, 23, 24] and the performance results of [35], we decided to use AES/Rijndael with GCM as the mode of operation as the basic AEAD algorithm for this work.

4.1.3.3 Data Classification & Monitoring

To achieve a better placement prediction of new data objects, we design a component that classifies each data object based on its object size. This classification process can be treated as a kind of grouping mechanism, whereby each data object belongs to exactly one class (e.g., size-class). In addition, the component should monitor the class-level access information of each data object. This information will then be considered by the optimization approach to improve the placement prediction of each data object. The classification component can be enabled or disabled manually by the client, unless the Optimizer component is disabled. This implies that also the classification component is disabled automatically and has therefore no effect on the placement solution.

The part of the component which is responsible for capturing the access information of data objects on class-level monitors four important values: the amount of read & write requests and the amount of data objects stored on standard & long-term storages. These four values are important for the result of the Classification component, which finally influences the placement result of the Optimizer.

4.2 System Model

This section defines the system model and all its relevant terms and variables required for the optimization approaches. Since our optimization approach is based on the work of J. Matt et al. [10], we can design our system model almost exactly like the announced global exact optimization approach of J. Matt et al. [10].

4.2.1 Variables

We declare the set of available cloud storages with S, whereby $s \in S = \{s_1, s_2, \ldots\}$ indicates a particular cloud storage. The set of all data objects can be defined as O, where $o \in O = \{o_1, o_2, \ldots\}$ represents one data object. Because we use Erasure Coding, a data object o is split into several data object fragments. The set of all data object fragments, which belong to exactly one data object o, can be determined as F_o and $f \in F_o = \{f_{o1}, f_{o2}, \ldots\}$ indicates a specific data object fragment, whereby $|S| \geq |F_o|$ always holds. As we announced in Section 2.5, the amount of data fragments can vary depending on the applied erasure code configuration. For example, if we apply an erasure code configuration of $EC(m, n) = EC(3, 4)$, the original data object is split into $m = 3$ data fragments, then encoded into a total of $n = 4$ fragments and afterwards stored on the most suitable cloud storage provider found by the optimization algorithm.

Most of the cloud storage provider offer a BRP model. We incorporate the different pricing models into the optimization approaches to realistically calculate the overall cost as accurately as possible. We can define B_s^{st} as the set of all price steps of the storage cost of storage s, whereas $b_s = (b_s^L, b_s^U, p_s) \in B_s^{st}$ defines one particular price step. The variable p_s holds the current price of the BRP model circumscribed by the two boundaries b_s^L and b_s^U, which define the lower and the upper bound of the price step b_s. The traffic cost B_{Tout} can be described analogously to the storage cost b_{st}. Since almost none of the storage provider charges for incoming data, we can omit the definitions of the cost calculation of incoming data transfer.

Every access to a data fragment is monitored and the resulting information is then stored in the Database of CORA. This historical information is used to predict the future access of each data fragment. We can define τ as the parameter which determines the amount of historical data used by optimization. In more detail, τ states how many minutes of historical data from now to the past is used.

4.2.2 Cost Model

Here, we define the several equations to calculate the cost which can occur when a data object is stored on a cloud storage s, gets requested from a cloud storage s, is transferred from a cloud storage s, or gets migrated from a cloud storage s_1 to a cloud storage s_2.

4.2.2.1 Overall Cost

Equation 4.1 calculates the overall cost of a data object fragment f, stored on a cloud storage s by taking into account the usage history of the past τ minutes.

$$c_{(s,f,\tau)} = c^{S}_{(s,f,\tau)} + c^{R}_{(s,f,\tau)} + c^{W}_{(s,f,\tau)} + c^{T_{in}}_{(s,f,\tau)} + c^{T_{out}}_{(s,f,\tau)} \tag{4.1}$$

4.2.2.2 Storage Cost

The first term $c^{S}_{(s,f,\tau)}$ of the overall cost calculation in (4.1) describes the storage cost. Equation 4.2 defines the cost that are billed respectively to the amount of space which a data object fragment f reserves on a cloud storage s.

$$c^{S}_{(s,f,\tau)} = p^{S}_{(s,\gamma_{(s,f)})} \cdot (\sigma_{(f,\tau)} + \hat{\sigma}_{(f,BTU)} \cdot h_f) \tag{4.2}$$

In more detail, the term $p^{S}_{(s,\gamma_{(s,f)})}$ calculates the storage price of a data object f on the cloud storage s. Most of the cloud storage provider offer different pricing model in respect to a predefined range of the used storage space consumption. To take such cost changes (i.e., cost-steps of a pricing model) into account, we define the parameter $\gamma_{(s,f)}$ which considers the used space of the current billing period of the data object fragment f stored on the cloud storage s. A data object fragment f, which is currently not stored on cloud storage s, is added to the calculation of $\gamma_{(s,f)}$.

The storage price $p^{S}_{(s,\gamma_{(s,f)})}$ is then multiplied with the size of the data object fragment f, defined as $\sigma_{(f,\tau)}$ by considering the past τ minutes of historical information of f. If the storage s has a defined BSU, and the size of the data object fragment f is smaller than this BSU, then the value of the BSU is used.

Additionally, if a data object f is already located on a long-term storage and the current BTU is not expired, the remaining BTU cost have to be included in the calculation of the overall storage cost. This can be achieved by the multiplication $\hat{\sigma}_{(f,BTU)} \cdot h_f$. The first term of the multiplication, $\hat{\sigma}_{(f,BTU)}$, defines the size of the data object fragment f that is charged for the remaining time till the end of the current BTU. The second term of the multiplication, the binary variable h_f, whereby $h_f \in \{1, 0\}$ always holds, indicates if a data object fragment f is currently stored on a long-term storage (i.e, $h_f = 1$) or not (i.e, $h_f = 0$).

4.2.2.3 Request Cost

The cost calculation for a read request is defined in (4.3). The term $r^R_{(f,\tau)}$ stands for the total number of performed read requests on a single data object fragment f in the last τ minutes.

$$c^R_{(s,f,\tau)} = r^R_{(f,\tau)} \cdot p^R_s \tag{4.3}$$

Analogue to (4.3), Equation 4.4 calculates the cost that occur on a write request. The term $r^W_{(f,\tau)}$ stands for the total number of performed write operations of a single data object fragment f in the last τ minutes.

$$c^W_{(s,f,\tau)} = r^W_{(f,\tau)} \cdot p^W_s \tag{4.4}$$

The delete request cost are calculated similar to (4.3) and (4.4). However, most of the cloud storage provider do not charge delete requests. As a result, we can neglect the delete request cost in our calculations.

4.2.2.4 Transfer Cost

Equation 4.5 defines the outgoing data transfer cost of a data object fragment f from a cloud storage s in the last τ minutes. The term $t^{out}_{(f,\tau)}$ represents the amount of bytes which were read from (i.e, outgoing data) the cloud storage s in the past time frame of τ. The outgoing transfer price of a data object fragment is labeled as $p^{T_{out}}_{(s,\beta_{(s,f)})}$. Similar to the storage cost in Section 4.2.2.2, most of the cloud provider charge lower prices in respect to the amount of transferred data. Therefore, analogue to $\gamma_{(s,f)}$ we can define $\beta(s,f)$, which calculates the amount of transferred bytes from the cloud storage s in the latest billing period.

Further, we have to consider if the data object fragment f is currently stored on a long-term storage s or not. In the first case, we have to charge additional retrieval cost. $p^{ret}_{(s,\beta_{(s,f)})}$ stands for the retrieval price, while the binary variable h_f can be equated to the term in Equation 4.2.

$$c^{T_{out}}_{(s,f,\tau)} = t^{out}_{(f,\tau)} \cdot (p^{T_{out}}_{(s,\beta_{(s,f)})} + p^{ret}_{(s,\beta_{(s,f)})} \cdot h_f) \tag{4.5}$$

The cost calculation for an incoming data transfer of a data object fragment f to a cloud storage s is defined in (4.6). This equation is similar to (4.5), with the difference that we do not have to charge retrieval cost $p^{ret}_{(s,\beta_{(s,f)})}$.

$$c^{T_{in}}_{(s,f,\tau)} = t^{in}_{(f,\tau)} \cdot p^{T_{in}}_{(s,\beta_{(s,f)})} \tag{4.6}$$

4.2.2.5 Migration Cost

If a data object fragment f has to be moved from one storage to another, we have to consider the originated migration cost. Basically, we can distinguish between two different cases for calculating migration cost.

1. In the first case, if the storage providers of the source and destination are different, the migration cost compounds of the outgoing cost of source storage s_1 and the incoming cost of destination storage s_2. Equation (4.7) defines this cost, whereas $p^{T_{out}}_{(s,\beta_{(s,f)})}$ and respectively $p^{T_{in}}_{(s,\beta_{(s,f)})}$, are equal to (4.5) and (4.6). If the source storage provider s_1 is a long-term storage, we have to consider additional data retrieval cost, defined by the multiplication of $p^{ret}_{s_1} \cdot h_f$, equally to (4.5). This migration price is then multiplied by $\hat{\sigma}_f$, which specifies the current size of the data object fragment f. Finally, we have to include the cost that occur related to the amount of read and write operations. This is done by the terms $r^R_{s_1} \cdot p^R_{s_1} + r^W_{s_2} \cdot p^W_{s_2}$ as in (4.3) and (4.4).

$$c^M_{(s_1,s_2,f)} = (p^{T_{out}}_{(s_1,\beta_{(s_1,f)})} + p^{T_{in}}_{(s_2,\beta_{(s_2,f)})} + p^{ret}_{s_1} \cdot h_f) \cdot \hat{\sigma}_f + r^R_{s_1} \cdot p^R_{s_1} + r^W_{s_2} \cdot p^W_{s_2} \quad (4.7)$$

2. In the other case, if the source and destination storage provider are identical, we have to consider a price reduction since most of the cloud storage providers charge lower prices when migrating data within the same storage provider. The relevant cost calculation is shown in (4.8). Similar to (4.7), the term $p^{T_{(out,red)}}_{(s_1,\beta_{(s_1,f)})}$ stands for the reduced outgoing migration price and $p^{T_{(in,red)}}_{(s_2,\beta_{(s_2,f)})}$ for the reduced incoming migration price, respectively between two regions within the same storage provider. The other remaining terms are the same as in (4.7).

$$c^{M_{red}}_{(s_1,s_2,f)} = (p^{T_{(out,red)}}_{(s_1,\beta_{(s_1,f)})} + p^{T_{(in,red)}}_{(s_2,\beta_{(s_2,f)})} + p^{ret}_{s_1} \cdot h_f) \cdot \hat{\sigma}_f + r^R_{s_1} \cdot p^R_{s_1} + r^W_{s_2} \cdot p^W_{s_2} \quad (4.8)$$

4.2.2.6 Decision Variables

To indicate if a data object fragment f is stored on a cloud storage s, we employ the binary decision variable $x_{(s,f)} \in \{0,1\}$, whereas $x_{(s,f)} = 1$ means that f is stored on s, and otherwise $x_{(s,f)} = 0$ if not. We use another binary decision variable $h_f \in \{0,1\}$ to mark if a data object fragment f is currently stored on a long-term storage, then $h_f = 1$, otherwise $h_f = 0$. Further, we define the decision variable $\hat{h}_s \in \{0,1\}$, which denotes if the cloud storage s is a long-term storage. $\hat{h}_s = 1$ if s is a long-term storage or contrary, $\hat{h}_s = 0$ if s is a standard storage. We define $z_{(s_1,s_2)} \in \{0,1\}$ as the decision variable that indicates if two cloud storages s_1 and s_2 are different storages or owned by different cloud storage providers. If this is true, then $z_{(s_1,s_2)} = 1$, otherwise $z_{(s_1,s_2)} = 0$. Analogous

to $z_{(s_1,s_2)} = 1$, $y_{(s_1,s_2)} \in \{0,1\}$ defines if two cloud storages s_1 and s_2 are different but appropriated within the same cloud storage provider. If this statement is true, then $y_{(s_1,s_2)} = 1$, otherwise $y_{(s_1,s_2)} = 0$.

Moreover, we have to define $g_{(\tilde{S},o)} \in \{0,1\}$, whereby $\tilde{S} = \{s_1, s_2, \ldots, s_n\}$, $|\tilde{S}| = |O|$ and $\tilde{S} \subseteq S$ always holds, as the set of selected cloud storages. $g_{(\tilde{S},o)} = 1$ indicates that each cloud storage $s \in \tilde{S}$ has exactly one data object fragment $f \in F_o$ stored. Otherwise, $g_{(\tilde{S},o)} = 0$ if at least one of the storages $s \in \tilde{S}$ does not have a data object fragment $f \in F_o$ stored.

To determine if the total used storage space of a cloud storage s is between an upper and a lower bound of the underlying Block Rate Pricing Model, we use the decision variables $u^{st}_{(s,b_s)} \in \{0,1\}$, $v^{st}_{(s,b_s)} \in \{0,1\}$ and $o^{st}_{(s,b_s)} \in \{0,1\}$. Referring to the first term more precisely, if the current used storage space of the cloud storage s is greater than the lower bound b^L_s of the current price step $b_s \in B^{st}_s$, then $u^{st}_{(s,b_s)} = 1$, otherwise $u^{st}_{(s,b_s)} = 0$. If the second term $o^{st}_{(s,b_s)} = 1$, this indicates that the current used storage space of the cloud storage s is lower than the upper bound b^U_s of the current price step $b_s \in B^{st}_s$, otherwise $o^{st}_{(s,b_s)} = 0$. Finally, the last term $o^{st}_{(s,b_s)}$ denotes if the current used storage space of the cloud storage s is between the lower bound b^L_s and the upper bound b^U_s of the current price step $b_s \in B^{st}_s$. This statement is true, if $o^{st}_{(s,b_s)} = 1$, otherwise $o^{st}_{(s,b_s)} = 0$. Furthermore, $o^{st}_{(s,b_s)} = 1$ implies that $u^{st}_{(s,b_s)} = 1$ and $v^{st}_{(s,b_s)} = 1$ always holds.

Analogously, we define the decision variables $u^{T_{out}}_{(s,b_s)} \in \{0,1\}$, $v^{T_{out}}_{(s,b_s)} \in \{0,1\}$ and $o^{T_{out}}_{(s,b_s)} \in \{0,1\}$.

4.3 Exact Global Optimization Approach

In this section, we will define and formulate the exact global optimization approach. We want to solve the problem of placing data objects on cloud storages with the goal of minimizing total storage cost while taking into account predefined QoS constraints.

4.3.1 Objective Function

The objective function of the exact global optimization problem, which minimizes the overall cost in global scope, is defined in (4.9).

$$\begin{aligned} \min \sum_{s \in S} \Bigg(\sum_{o \in O} \sum_{f \in F_o} \; & \big(c^R_{(s,f,\tau)} \cdot c^W_{(s,f,\tau)} + c^M_{(\hat{s}_f,s,f)} \cdot z_{(\hat{s}_f,s)} \\ & + c^{M_{red}}_{(\hat{s}_f,s,f)} \cdot y_{(\hat{s}_f,s)} \big) \cdot x_{(s,f)} \cdot l_{(s,f)} \Bigg) + c^{st}_{(s,\tau)} + c^{T_{out}}_{(s,\tau)} \end{aligned} \tag{4.9}$$

We will now analyze the objective function in more detail. As shown in (4.9), the objective function is a triple-sum which iterates over all cloud storages $s \in S$, over all data objects $o \in O$ and finally, over all data object fragments $f \in F_o$.

As has been discussed in Section 4.2.2.3, the first two terms are the request cost. The first term, $c^R_{(s,f,\tau)}$ represents the read request cost as defined in (4.3). The second term, $c^W_{(s,f,\tau)}$ indicates the write request cost, defined in (4.4) respectively. The succeeding terms are $c^M_{(\hat{s}_f,s,f)} \cdot z_{(\hat{s}_f,s)}$ and $c^{M_{red}}_{(\hat{s}_f,s,f)} \cdot y_{(\hat{s}_f,s)}$, which represent the migration cost, mentioned in Section 4.2.2.5 and defined in (4.7) and (4.8). Finally, this first major block, generally composed of request and migration cost, only affects the objective function if the current data object fragment f is stored on cloud storage s. Therefore, this block is multiplied with the appropriate decision variable $x_{(s,f)}$, defined in Section 4.2.2.6, and the penalty factor $l_{(s,f)}$, which is defined in (4.22).

4.3.2 Cost Constraints

The remaining terms of the overall cost calculation, defined in (4.9), are the storage cost $c^{st}_{(s,\tau)}$ and the outgoing data transfer cost $c^{T_{out}}_{(s,\tau)}$.

Storage Cost $c^{st}_{(s,\tau)}$

The following Equations 4.10 to 4.14, show if the amount of storage space used is in the range of a specific price step of the underlying storage pricing model of a cloud storage s.

The equations shown in (4.10) and (4.11) define, if the total amount of used storage space on a cloud storage s is greater than the lower bound b^L_s of a specific price step $b_s \in B^{st}_s$. If this statement is true, then $u^{st}_{(s,b_s)} = 1$, otherwise $u^{st}_{(s,b_s)} = 0$. Furthermore, we define M as the constant which holds the largest possible value of $\sigma_{(f,\tau)}$, at least b^L_s or b^U_s.

$$\forall s \in S,\ \forall b_s \in B^{st}_s,\ \exists b^L_s \in b_s\colon\quad b^L_s \leq \sum_{o \in O} \sum_{f \in F_o} \sigma_{(f,\tau)} \cdot x_{(s,f)} \ +\ M \cdot (1 - u^{st}_{(s,b_s)}) \tag{4.10}$$

$$\forall s \in S,\ \forall b_s \in B^{st}_s,\ \exists b^L_s \in b_s\colon\quad b^L_s > \sum_{o \in O} \sum_{f \in F_o} \sigma_{(f,\tau)} \cdot x_{(s,f)} \ +\ M \cdot u^{st}_{(s,b_s)} \tag{4.11}$$

Conversely to (4.10) and (4.11), the equations shown in (4.12) and (4.13) define, if the total amount of used storage space on a cloud storage s is lower than the upper bound b^U_s of a specific price step $b_s \in B^{st}_s$. If this statement is true, then $v^{st}_{(s,b_s)} = 1$, otherwise $v^{st}_{(s,b_s)} = 0$.

$$\forall s \in S,\ \forall b_s \in B^{st}_s,\ \exists b^U_s \in b_s\colon\quad b^U_s \leq \sum_{o \in O} \sum_{f \in F_o} \sigma_{(f,\tau)} \cdot x_{(s,f)} \ +\ M \cdot (1 - u^{st}_{(s,b_s)}) \tag{4.12}$$

$$\forall s \in S,\ \forall b_s \in B^{st}_s,\ \exists b^U_s \in b_s\colon\quad b^U_s > \sum_{o \in O} \sum_{f \in F_o} \sigma_{(f,\tau)} \cdot x_{(s,f)} \ +\ M \cdot u^{st}_{(s,b_s)} \tag{4.13}$$

Further, (4.14) indicates if the current used storage space of a cloud storage s is between the lower bound b_s^L and the upper bound b_s^U. If this statement holds true and due to the definitions in (4.10) to (4.13), it follows that $u^{st}_{(s,b_s)} = 1$ and $v^{st}_{(s,b_s)} = 1$, which implies again that $o^{st}_{(s,b_s)} = 1$. Otherwise, $o^{st}_{(s,b_s)} = 0$ applies, which also means that at least one of the decision variables $u^{st}_{(s,b_s)}$ or $v^{st}_{(s,b_s)}$ is false.

$$\forall s \in S,\ \forall b_s \in B_s^{st}\colon\ 0 \leq u^{st}_{(s,b_s)} + v^{st}_{(s,b_s)} - 2 \cdot o^{st}_{(s,b_s)} \leq 1 \tag{4.14}$$

Finally, we can define the storage cost that are billed based on the current used storage space of a cloud storage s by considering the current price p_s of a specific price step $b_s \in B_s^{st}$, shown in (4.15). M is the constant which represents the largest possible value of at least $\sigma_{(f,\tau)} + \hat{\sigma}_{(f,\tau)} \cdot (h_f + \hat{h_s})$ times the largest possible value of p_s, and further at least the largest possible value of c_s^{st}.

$$\begin{aligned}
&\forall s \in S,\ \forall b_s \in B_s^{st},\ \exists p_s \in b_s\colon \\
&\sum_{o \in O} \sum_{f \in F_o} p_s \cdot \left(\sigma_{(f,\tau)} + \hat{\sigma}_{(f,\tau)} \cdot (h_f + \hat{h_s})\right) \cdot x_{(s,f)} - M \cdot (1 - o^{st}_{(s,b_s)}) \leq c_s^{st} \\
&\leq \sum_{o \in O} \sum_{f \in F_o} p_s \cdot \left(\sigma_{(f,\tau)} + \hat{\sigma}_{(f,\tau)} \cdot (h_f + \hat{h_s})\right) \cdot x_{(s,f)} + M \cdot (1 - o^{st}_{(s,b_s)})
\end{aligned} \tag{4.15}$$

Transfer Cost $c^{T_{out}}_{(s,\tau)}$

The transfer cost can be analogously defined as the already mentioned calculations of Storage Cost $c^{st}_{(s,\tau)}$. Equations (4.16) to (4.20), show if the amount of bytes of the outgoing data is in the range of a specific price step of the underlying transfer pricing model of a cloud storage s.

The equations shown in (4.16) and (4.17) define, if the total amount of bytes of the outgoing data on a cloud storage s is greater than the lower bound b_s^L of a specific price step $b_s \in B_s^{T_{out}}$. If this statement is true, then $u^{T_{out}}_{(s,b_s)} = 1$, otherwise $u^{T_{out}}_{(s,b_s)} = 0$. Furthermore, we define M as the constant which holds the largest possible value of $t^{out}_{(f,\tau)}$, at least b_s^L or b_s^U.

$$\forall s \in S,\ \forall b_s \in B_s^{T_{out}},\ \exists b_s^L \in b_s\colon\ \ b_s^L \leq \sum_{o \in O} \sum_{f \in F_o} t^{out}_{(f,\tau)} \cdot x_{(s,f)} \ +\ M \cdot (1 - u^{T_{out}}_{(s,b_s)}) \tag{4.16}$$

$$\forall s \in S,\ \forall b_s \in B_s^{T_{out}},\ \exists b_s^L \in b_s\colon\ \ b_s^L > \sum_{o \in O} \sum_{f \in F_o} t^{out}_{(f,\tau)} \cdot x_{(s,f)} \ +\ M \cdot u^{T_{out}}_{(s,b_s)} \tag{4.17}$$

Conversely to (4.16) and (4.17), the equations shown in (4.18) and (4.19) define, if the total size of outgoing data on a cloud storage s is lower than the upper bound b_s^U of

a specific price step $b_s \in B_s^{T_{out}}$. If this statement is true, then $v_{(s,b_s)}^{T_{out}} = 1$, otherwise $v_{(s,b_s)}^{T_{out}} = 0$.

$$\forall s \in S,\ \forall b_s \in B_s^{T_{out}},\ \exists b_s^U \in b_s\colon\ b_s^U \leq \sum_{o \in O} \sum_{f \in F_o} t_{(f,\tau)}^{out} \cdot x_{(s,f)} \ +\ M \cdot (1 - u_{(s,b_s)}^{T_{out}}) \tag{4.18}$$

$$\forall s \in S,\ \forall b_s \in B_s^{T_{out}},\ \exists b_s^U \in b_s\colon\ b_s^U > \sum_{o \in O} \sum_{f \in F_o} t_{(f,\tau)}^{out} \cdot x_{(s,f)} \ +\ M \cdot u_{(s,b_s)}^{T_{out}} \tag{4.19}$$

Further, (4.20) indicates if the current amount of bytes of outgoing data from a cloud storage s is between the lower bound b_s^L and the upper bound b_s^U. If this statement holds true and due to the definitions in (4.16) to (4.19), it follows that $u_{(s,b_s)}^{T_{out}} = 1$ and $v_{(s,b_s)}^{T_{out}} = 1$ which implies again that $o_{(s,b_s)}^{T_{out}} = 1$. Otherwise, $o_{(s,b_s)}^{T_{out}} = 0$, resulting that at least one of the decision variables $u_{(s,b_s)}^{T_{out}}$ or $v_{(s,b_s)}^{T_{out}}$ holds false.

$$\forall s \in S,\ \forall b_s \in B_s^{st}\colon\ 0 \leq u_{(s,b_s)}^{T_{out}} + v_{(s,b_s)}^{T_{out}} - 2 \cdot o_{(s,b_s)}^{T_{out}} \leq 1 \tag{4.20}$$

Finally, we can define the data transfer cost that are billed based on the total size of transferred bytes of a cloud storage s taking into account current price p_s of a specific price step $b_s \in B_s^{T_{out}}$, shown in (4.21). M is the constant which holds the largest possible value of at least $(p_s + p_s^{ret} \cdot h_f)$ times the largest possible value of $t_{(f,\tau)}^{out}$, and further at least the largest possible value of $c_s^{T_{out}}$.

$$\begin{aligned}
&\forall s \in S,\ \forall b_s \in B_s^{T_{out}},\ \exists p_s \in b_s\colon \\
&\sum_{o \in O} \sum_{f \in F_o} t_{(f,\tau)}^{out} \cdot (p_s + p_s^{ret} \cdot h_f) \cdot x_{(s,f)} - M \cdot (1 - o_{(s,b_s)}^{T_{out}}) \leq c_s^{T_{out}} \\
&\leq \sum_{o \in O} \sum_{f \in F_o} t_{(f,\tau)}^{out} \cdot (p_s + p_s^{ret} \cdot h_f) \cdot x_{(s,f)} + M \cdot (1 - o_{(s,b_s)}^{T_{out}})
\end{aligned} \tag{4.21}$$

Penalty Factor $l_{(s,f)}$

Equation (4.22) defines the penalty factor $l_{(s,f)}$ of a storage s, a data object fragment f, and the decision variable $\hat{h_f}$ as described in Section 4.2.2.6. Here, $l_{(s,f)}^{cl}$ defines the probability value on class-level with $0 \leq l_{(s,f)}^{cl} \leq 1$ that a data object fragment f will be stored on a standard storage s.

$$l_{(s,f)} = 1 + l_{(s,f)}^{cl} \cdot \hat{h_f} \tag{4.22}$$

Based on the monitored data object fragment distribution between long-term and standard storages, combined with the result of the classification component, the penalty factor

$l_{(s,f)}$ can take values between 1 and 2. For example, if a data object is classified to be stored on a standard storage with a probability value of $l^{cl}_{(s,f)} = 80\%$, the penalty factor $l_{(s,f)} = 1 + l^{cl}_{(s,f)} = 1.8$ reduces the chance (i.e., increases the total price by the factor 1.8) that the data object will be stored on long-term storages.

Boundary Definitions

To make sure that our optimization yields correct results we have to define boundary constraints as shown in (4.23).

$$\begin{array}{lll} c_s^{st} \geq 0 & c_s^{T_{out}} \geq 0 & p_{(s,f)} \in \{1,2\} \\ u^{st}_{(s,b_s)} \in \{0,1\} & v^{st}_{(s,b_s)} \in \{0,1\} & o^{st}_{(s,b_s)} \in \{0,1\} \\ u^{T_{out}}_{(s,b_s)} \in \{0,1\} & v^{T_{out}}_{(s,b_s)} \in \{0,1\} & o^{T_{out}}_{(s,b_s)} \in \{0,1\} \end{array} \tag{4.23}$$

4.3.3 QoS Constraints

In this section, we define our QoS constraints which are fundamental for the optimization process. These constraints can be predefined by the user as so-called SLOs.

Vendor Lock-In Factor

The constraint in (4.24) or in greater detail, the defined user SLO, ensures that the vendor lock-in factor is always fulfilled during the optimization.

$$\forall o \in O\colon \ \frac{1}{\sum_{s \in S} \sum_{f \in F_o} x_{(s,f)}} \leq l_o \tag{4.24}$$

Availability

To guarantee, for every data object $o \in O$, a specific user-defined minimum level of availability, we determine the constraint shown in (4.25).

$$\forall o \in O\colon \ \sum_{i=m}^{n} \sum_{\tilde{S}' \in [\tilde{S}]'} \Big(\prod_{s \in \tilde{S}'} a_s \cdot \prod_{s \in \tilde{S} \setminus \tilde{S}'} (1 - a_s) \Big) \geq a_o \cdot g_{(\tilde{S},o)} \tag{4.25}$$

We can define $\tilde{S} \subseteq S$, as a specific subset of S which is selected by the optimization algorithm to store $f \in F_o$ data object fragments on $s \in \tilde{S}$ cloud storages, whereby $\tilde{S} = \{s_1, s_2, \ldots, s_n\}$ and $|\tilde{S}| = |F_o| = n$ holds. As mentioned in Section 2.5, each data object $o \in O$ is split into the respective amount of n data object fragments $f \in F_o$ and finally stored on the best fitting cloud storages $s \in \tilde{S}$. $[\tilde{S}]^i = \{R \subseteq \tilde{S}, |R| = i\}$ represents the set of possible i-subsets of $\tilde{S}$ and $\tilde{S}'$ holds all i-combinations of the set $\tilde{S}$, respectively. The term a_s indicates the availability of a cloud storage s.

The inner sum $\sum_{\tilde{S}' \in [\tilde{S}]'} \left(\prod_{s \in \tilde{S}'} a_s \cdot \prod_{s \in \tilde{S} \setminus \tilde{S}'} (1 - a_s) \right)$, defined in Equation 4.25, calculates the availability of the subset of $\tilde{S}'$.

Further, we have to pay attention to the erasure code configuration applied to our system. The erasure code configuration $EC(m, n)$, as explained in Section 2.5 in more detail, can tolerate a simultaneous malfunction of a maximum of $(n - m)$ cloud storages. This effects the availability of our system. Therefore, we have to sum up the calculated availability of the storage set $\tilde{S}$ (i.e., from $i = m$ up to $n = |\tilde{S}|$), achieved by the outer sum $\sum_{i=m}^{n}$.

Finally, the calculated availability is compared with $a_o \cdot g_{(\tilde{S},o)}$, whereby a_o represents the minimum required user-defined availability of a data object o. $g_{(\tilde{S},o)}$, as has already been mentioned in Section 4.2.2.6, states if each cloud storage $s \in \tilde{S}$ has exactly one data fragment $f \in F_o$ stored or not.

Durability

To accomplish a user-defined minimum level of durability, we define for each data object $o \in O$ the constraint specified in (4.26). This constraint can be defined equally as in (4.25), with the only difference of the variables d_s and d_o. d_s indicates the durability of a cloud storage s and d_o holds the user-defined minimum required durability of a data object $o \in O$, analogues to a_s and a_o in (4.25).

$$\forall o \in O\colon \sum_{i=m}^{n} \sum_{\tilde{S}' \in [\tilde{S}]'} \Big(\prod_{s \in \tilde{S}'} d_s \cdot \prod_{s \in \tilde{S} \setminus \tilde{S}'} (1 - d_s) \Big) \geq d_o \cdot g_{(\tilde{S},o)} \tag{4.26}$$

4.3.4 Further Constraints

In this section we define the remaining constraints which are important for our optimization algorithm.

As we have already discussed in Section 4.1.2.3, CORA, the used middleware of our system requires to read only m fragments in case of downloading a data object o. Because cloud storage providers usually do not charge any cost for uploading data objects (i.e., incoming data of a cloud storage) and long-term storages are in terms of storage cost significantly cheaper than standard storages, we can always store $(n - m)$ data object fragments on long-term storages to lower the total cost. The constraint shown in (4.27) ensures that $(n - m)$ data object fragments are stored on long-term storages, whereas $\hat{h}_s$ and $x_{(s,f)}$ are the decision variables mentioned in Section 4.2.2.6. The former indicates if a cloud storage s is a long-term storage or not, the latter states if a data object fragment f is stored on a cloud storage s or not.

$$\forall o \in O\colon \sum_{s \in S} \sum_{f \in F_o} x_{(s,f)} \cdot \hat{h}_s \geq n - m \tag{4.27}$$

Constraints 4.28 and 4.29 define if each cloud storage $s \in \tilde{S}$ has exactly one data object fragment $f \in F_o$ stored (i.e., $g_{(\tilde{S},F)} = 1$) or not (i.e., $g_{(\tilde{S},F)} = 0$). If we combine both (4.28) and (4.29), the resulting functionality can be treated like a conjunction.

$$\forall o \in O\colon\ g_{(\tilde{S},o)} \geq \sum_{s \in \tilde{S}} \sum_{f \in F_o} x_{(s,f)} - (n-1) \tag{4.28}$$

$$\forall o \in O, \forall s \in \tilde{S}\colon\ g_{(\tilde{S},o)} \leq \sum_{f \in F_o} x_{(s,f)} \tag{4.29}$$

If a data object $o \in O$ is split into the corresponding data object fragments $f \in F_o$ and then distributed to the different cloud storages, it could be the case that multiple data object fragments $f \in F_o$ of the same data object $o \in O$ get assigned to the same cloud storage $s \in S$. To prevent such a special case, we define Constraint 4.30, which ensures that only n assignments from data object fragments $f \in F_o$ to the storages $s \in S$ exist.

$$\forall o \in O\colon\ \sum_{s \in S} \sum_{f \in F_o} x_{(s,f)} = n \tag{4.30}$$

Conversely, it could be the case that one particular data object fragment $f \in F_o$ gets assigned to multiple different cloud storages in S. To prevent this, we have to define three more constraints as shown in (4.31), (4.32) and (4.33).

The first constraint, defined in (4.31), guarantees that a data object fragment f can only be assigned once to a specific cloud storage s. The second constraint in (4.32), ensures that a cloud storage s has a maximum of one data object fragment $f \in F_o$, of a specific data object o, assigned. In this context, the last constraint in (4.33) guarantees that $x_{(s,f)}$ does not hold a negative value.

$$\forall o \in O, \forall f \in F_o\colon\ \sum_{s \in S} x_{(s,f)} = 1 \tag{4.31}$$

$$\forall o \in O, \forall s \in S\colon\ \sum_{f \in F_o} x_{(s,f)} \leq 1 \tag{4.32}$$

$$\forall o \in O, \forall s \in S\colon\ \sum_{f \in F_o} x_{(s,f)} \geq 0 \tag{4.33}$$

Finally, in (4.34) we define that our used decision variables are restricted to the range of Boolean values $\{0, 1\}$.

$$\begin{aligned} g_{(\tilde{S},o)} &\in \{0,1\} \qquad z_{(s_1,s_2)} \in \{0,1\} \\ x_{(s,f)} &\in \{0,1\} \qquad y_{(s_1,s_2)} \in \{0,1\} \end{aligned} \tag{4.34}$$

4.4 Global Heuristic Optimization Approach

In this section, we introduce the global heuristic optimization approach which will find a near-optimal placement solution faster than the exact global optimization approach.

4.4.1 Requirements

The global heuristic approach is designed to solve optimization problems with reference to a much more practical orientation. The approach ensures finding a feasible and near-optimal solution of the problem instance. As we have already described in Section 2.3.3, the effort of the exact global optimization approach increases exponentially with the problem size. In our particular case, this means that the problem size is equivalent to the amount of data objects together with the amount of used cloud storages. As our optimization approaches are designed to solve the placement problem in a global scope, the optimization approach has to include all data objects in the calculation. As a result, the complexity and resource consumption of the overall system gets very high. This makes the exact global optimization approach not applicable in real-world scenarios where big amounts of data need to be processed in a reasonable time. This is why we designed a global heuristic approach that aims at finding a cost-efficient placement solution of the data objects as fast as possible, but nevertheless a solution which is qualitatively close enough compared to the exact global optimization approach. The heuristic approach is therefore well-suited for a real-world scenario where big amounts of data objects must be processed.

4.4.2 Algorithm

Global Heuristic Placement Function

The simplistic pseudo code of the global heuristic placement algorithm is shown in 4.1. For each request that is performed on a data object, the Optimizer of SCORA calls the placement function *getOptimizedProviders* with its passed input parameters *allDataObjects* and *dataObject*. Especially, in case of an UPLOAD request (as shown in line 5) and if the data object has not already been monitored by the middleware (i.e., *dataObject* is a new data object), the placement function *getPlacementsForNewDataObject* is called. For every other request type (e.g., DOWNLOAD, UPDATE, DELETE), the placement function *getPlacementsForUnusedDataObjects* is called to find rarely used or unused data objects that can be migrated from standard storages to long-term storages.

Placement Function For New Data Object

The pseudo code of the function *getPlacementsForNewDataObject* as mentioned in Algorithm 4.1, Line 5, is shown in 4.2 more precisely. Each time an UPLOAD command is received by the Optimizer, the function *getPlacementsForNewDataObject* including the parameter *dataObject* (e.g., the data object which is uploaded) is called.

Algorithm 4.1: Global Heuristic Placement Function

Input : *allDataObjects* - The set of all previously monitored data objects
dataObject - The data object to upload
Output : *placements* - A list of assignments from cloud storages $s \in S$ to data object fragments $f \in F$

```
1  Function getOptimizedProviders(allDataObjects, dataObject):
2      for each request do
3          placements ← {∅};
4          if requestType == UPLOAD then
5              placements ← getPlacementsForNewDataObject(dataObject);
6          else
7              placements ←
                getPlacementsForUnusedDataObjects(allDataObjects);
8          end
9      end
10     return placements;
11 end
```

Recap We use erasure coding $EC(m, n)$ as redundancy mechanism which splits a data object in n commensurate fragments. As we already have mentioned in Section 2.5, we only need to read m fragments to reconstruct the original data object. As a result, our placement optimization approach can basically store $n - m$ data object fragments on long-term storages. The remaining m fragments can be placed on the standard storages where the overall cost for storing and accessing the fragments is minimal.

In order to achieve a cost-efficient placement of the data object fragments, we create a function that ranks our used cloud storages set according to the cheapest price. To be more precisely, we create two functions, *getRankedStandardStorages* in Line 4, which returns a subset of ranked standard storages and *getRankedLongTermStorages* in Line 5, which returns a subset of ranked long-term storages of the overall set of cloud storages.

Most of the cloud storage providers have defined a Billing Time Unit (BTU) as well as a Billing Storage Unit (BSU), which will be described later in Section 5.1.1. Therefore, Lines 6 up to 10 are very important for the cost efficiency of our solution. This code block ensures that a data object fragment, whose file size is smaller than the BSU of a selected storage, is not stored on such a cloud storage (e.g., long-term storage). This is important because otherwise we would be paying fees for cloud storage space that is not used physically.

Lines 11 up to 15 consider the result of our classification component as introduced in Section 4.1.3.3. The classification component predicts the placement distribution

of the data object fragments between long-term and standard storages based access information that is monitored on class-level. If the classification is enabled, the function $getDataShardsAmountWithClassification$ is called which returns the amount of data object fragments that are predicted to be stored on standard storages. Otherwise, if the component is disabled, m data object fragments as defined by the erasure coding configuration $EC(m, n)$ are stored on standard storages.

Because of the Lines 6 up to 10 and the prediction of the value m generated by the classification component, shown in Lines 11 up to 15, it could be happen that the current available amount of long-term storages is less than the required amount of $n - m$. Therefore, it is important to consider this special case to correct the value of m by subtracting the amount of available long-term storages from the total amount of data object fragments n. This ensures that the set of data object fragments resulting from the optimization algorithm and to be stored on long-term storages is smaller than or equal to the available set of long-term storages. This is essential for the functional correctness of our optimization approach.

Due to the corrected result of the optimized value m, which represents the amount of data object fragments stored on standards storages, we can construct the optimized amount of data object fragments stored on long-term storages by performing the calculation of $n - m$. Now, we can select the m best ranked standard storages, shown in Lines 20 and 21. Analogously, we select the $n - m$ best-ranked long-term storages according to the storage ranking, which is shown in Lines 23 and 24. Finally, each of the n storages get assigned exactly one data object fragment of the data object *dataObject*, shown in Lines 26 and 27.

With this approach, we exploit the price advantages of both, standard and long-term storages. Attention has to be given when storing data object fragments on long-term storages. Since long-term storages are particularly designed for storing rarely-accessed data objects, the request cost are very high compared to standard storages. Therefore, the involved extra cost have to be carefully considered to achieve a cost reduction by using long-term storages.

Algorithm 4.2: Placement Function for a New Data Object

Input : *dataObject* - The data object to upload
Output : *placements* - A list of assignments from cloud storages $s \in S$ to data object fragments $f \in F$

```
1  Function getPlacementsForNewDataObject(dataObject):
2      placements ← {∅};
3      storages ← {∅};
4      standardStorages ← getRankedStandardStorages(dataObject);
5      longTermStorages ← getRankedLongTermStorages(dataObject);
6      foreach storage in longTermStorages do
7          if chunkSize(dataObject) < minBilledFileSize(storage) then
8              longTermStorages.remove(storage);
9          end
10     end
11     if classification is enabled then
12         m ← getDataShardsAmountWithClassification(dataObject);
13     else
14         m ← getDataShardsAmount(dataObject);
15     end
16     n ← getTotalShardsAmount(dataObject);
17     if (n − m) > count(longTermStorages) then
18         m ← n − count(longTermStorages);
19     end
20     for i ← 1 to m do
21         storages.add(standardStorages[i])
22     end
23     for i ← 1 to (n − m) do
24         storages.add(longTermStorages[i])
25     end
26     for i ← 1 to n do
27         placements.add(storages[i], dataObject.fragments[i])
28     end
29     return placements;
30 end
```

Placement Function For Unused Data Objects

The function shown in Algorithm 4.3 optimizes the placement of rarely or unused data objects. In order to achieve this, we use the monitored access history information of each data object, or more precisely, the access history information of each data object fragment. A data object fragment that is stored on a standard storage is considered unused if it is not accessed during the last time interval of the systems' predefined BTU. The process for the optimization of all unused data objects stored on standard storages is shown from Line 1 up to 21.

Algorithm 4.3: Placement Function for Unused Data Objects

Input : *allDataObjects* - A list of all data objects
Output : *placements* - An optimized list of assignments from cloud storages $s \in S$ to data object fragments $f \in F$

```
1  Function getPlacementsForUnusedDataObjects(allDataObjects):
2      placements ← {∅};
3      foreach dataObject ∈ allDataObjects do
4          F_dataObject ← dataObject.fragments;
5          foreach fragment ∈ F_dataObject do
6              if currentStorage(fragment) is a standard storage
7               & lastTimeAccessed(fragment) is longer than BTU then
8                  longTermStorages ← getRankedLongTermStorages(fragment);
9                  foreach f ∈ F_dataObject do
10                     if longTermStorages contains currentStorage(f) then
11                         longTermStorages.remove(currentStorage(f));
12                     end
13                 end
14                 if longTermStorages.length > 0 then
15                     placements.add(longTermStorages[0], fragment);
16                 end
17             end
18         end
19     end
20     return placements;
21 end
```

As shown in Line 3, we loop through the set of all data objects (e.g., *allDataObjects*) that are monitored by the middleware. Furthermore, for every data object (i.e., *dataObject*) of this set, we iterate over all its associated data object fragments as shown in Line 5. For each data object fragment, we first have to check if the data object fragment is currently stored on a standard storage, as shown in Line 6. Second, we need to check if the last

access time of the data object fragment was before the last time interval of size of the BTU. If both conditions are fulfilled, we can search for the best-ranked long-term storage, as shown in Line 8.

Since we want to avoid vendor lock-in and provide high availability, we have to exclude those long-term storages that are already assigned to the same data object. This important exclusion is shown in the Lines 9 up to 13. This ensures that a fragment $f \in F_o$ of a data object $o \in O$ is not stored on the same cloud storage $s \in S$. Finally, if long-term storages for the current data object fragment are available (Line 14), we can create a new placement solution by assigning the best ranked long-term storage to the current unused fragment, as shown in Line 15.

4.5 Implementation

4.5.1 Miscellaneous

As we already have mentioned in Section 4.1.2, we use CORA [9] as foundation for this work. CORA is written in the programming language Java[1] in the Version 8[2] (Java 8). Principally designed as a Web application with a Representational State Transfer (REST) API, CORA uses as starting point the Web framework Spring Boot[3] which provides many built-in and easy-to-use features for Web applications. Furthermore, the middleware uses Maven[4] as build management tool.

CORA can generally be used as a real-world cloud-based middleware. The system is designed loosely-coupled for an easy integration of different placement optimization approaches. This enables the possibility to implement several optimization approaches (e.g., local exact optimization, local heuristic optimization, global exact optimization, global heuristic optimization, etc.) and afterwards simply setting the desired approach to be executed.

To facilitate the testing and evaluation of these different optimization approaches against a real-world scenario, CORA provides an additional simulation wrapper to achieve this. If the simulation mode is activated, the middleware takes a file access trace as input parameter. This input file can usually be a simple text file (e.g., .csv) whereas in each line at least the access timestamp, file identifier, request type and file size has to be provided. CORA reads each line of the file access trace and transforms the commands into corresponding real-world API requests. To ensure the time correctness of each request, the middleware programmatically changes the system time to the time provided in the file access trace (e.g., access timestamp). In order to achieve this, CORA makes use of the time library called Joda Time[5].

[1] https://www.java.com/de/
[2] https://www.java.com/de/download/faq/java8.xml
[3] https://spring.io/projects/spring-boot
[4] https://www.apache.org
[5] https://www.joda.org/joda-time/

Another important feature provided by CORA is the use of erasure coding to split data objects into several commensurate fragments to provide the possibility of a dynamic data object placement on different cloud storages. This is done by using Backblaze[6] Reed-Solomon, an open source Java library which is utilized with a very efficient implementation of Read-Solomon erasure coding.

4.5.2 Compression Component

To reduce the required cloud storage space of each data object and thus, to minimize the overall cost, we make use of the Java Library Apache Commons Compress[7] in the release version 1.18[8]. This compression library provides multiple compression/decompression algorithms like tar, zip, gzip, XZ, Pack200, bzip2, 7z, lzma, DEFLATE, Brotli and much more. The library provides abstract base classes and Java factories for compressors and archivers that can be used to choose implementations by the name of algorithm. Furthermore, in case of reading compressed data objects, the library can guess the format from the input stream and automatically chooses the matching decompression implementation. As we decided to use gzip for our work (see Section 4.1.3.1), we statically initiated the compression library to only use this algorithm for our work.

An important feature, we have implemented additionally, is the check of the file size immediately after the compression process. Depending on the structure of the input byte stream, the compression output may result in a larger file size than the original file. Therefore, our compression component compares both values and chooses always the best (i.e., smallest) outcome.

In case of reading a data object, it is therefore essential to know if a data object is compressed or not. To achieve this, we store the compression format applied to a data object in the database. For example, if a data object o is stored compressed on the cloud storages, the associated format "gzip" is stored as string in the database to provide additional information for a data object o. If no compression is applied, the compression format is set to an empty string.

4.5.3 Encryption Component

To ensure confidentiality, integrity and authenticity of the overall system, we extended CORA by an additional encryption component. This is done by using an AEAD algorithm or in greater detail, applying AES with GCM as the mode of operation on each data object. Since Java v1.4 (Java 4) the library Java Cryptography Extension (JCE) is included which provides several ciphers, MACs, key management, secure objects and digital signatures.

AES operated in the GCM block mode provides all the requirements, is easy to use and is available in most Java environments. GCM is basically a Counter Mode (CTR) mode and

[6] https://www.backblaze.com/open-source-reed-solomon.html
[7] https://commons.apache.org/proper/commons-compress/
[8] https://commons.apache.org/proper/commons-compress/javadocs/api-1.18/index.html

calculates an authentication tag sequentially during encryption that is usually appended to the cipher text. As the size of the authentication tag is an important feature, it should be at least 128 bit long [36].

Now, we want to give an short overview of how the encryption process works. First, we generate a strong random 128 bit key as shown in Listing 4.1, Lines 4 up to 7.

Listing 4.1: Code Snippet - Secret Key Generation

```
1 ...
2 protected final static int KEY_SIZE = 128;
3 ...
4 SecureRandom random = SecureRandom.getInstanceStrong();
5 KeyGenerator keyGen = KeyGenerator.getInstance("AES");
6 keyGen.init(KEY_SIZE, random);
7 SecretKey secretKey = keyGen.generateKey();
8 ...
```

Next, we create an Initialization Vector (IV) with a 12-byte random byte array as recommend[9] by NIST when using GCM as block cipher mode of operation. The generation of the IV is shown in Listing 4.2, Lines 4 up to 6.

Listing 4.2: Code Snippet - IV Generation

```
1 ...
2 protected final static int IV_BYTES = 12;
3 ...
4 byte[] iv = new byte[IV_BYTES];
5 SecureRandom secRandom = new SecureRandom();
6 secRandom.nextBytes(iv);
7 ...
```

Finally, we initialize the cipher in AES-GCM mode and construct the cipher message as shown in Listing 4.3, Line 5. The complete cipher message is wrapped into one single byte array (see Listing 4.3, Lines 11-15) and consists of the length of the IV, the IV itself, the cipher text (i.e., the encrypted data) and the authentication tag whereas the latter is automatically appended to the cipher text by the encryption library.

[9] https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf

Listing 4.3: Code Snippet - Encryption Implementation

```
1  ...
2  protected final static int GCM_TAG_SIZE = 128;
3  ...
4  public byte[] encrypt(byte[] plainText, ...) {
5      final Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
6      GCMParameterSpec parameterSpec = new GCMParameterSpec(
           GCM_TAG_SIZE, iv); //128-bit auth tag length
7      SecretKeySpec secretKeySpec = new SecretKeySpec(secretKey, "AES")
           ;
8      cipher.init(Cipher.ENCRYPT_MODE, secretKeySpec, parameterSpec);
9      byte[] cipherText = cipher.doFinal(plainText);
10     ...
11     ByteBuffer byteBuffer = ByteBuffer.allocate(Integer.BYTES + iv.
           length + cipherText.length);
12     byteBuffer.putInt(iv.length);
13     byteBuffer.put(iv);
14     byteBuffer.put(cipherText);
15     byte[] cipherMessage = byteBuffer.array();
16     ...
17 }
```

We further describe the decryption process in Listing 4.4. First, the length of the transmitted IV is unwrapped from the cipher message by fetching the first four bytes (see Line 7), which represents the allocated space for an integer value in Java. As GCM only provides authentication on the cipher text, the IV length and its value are transmitted as raw text and can therefore be altered by a malicious party. Therefore, checking of the transmitted IV size length is important (see Lines 8-10). This value is then used to get the appended IV as shown in Line 12 and 13. Finally, the remaining bytes (i.e., the cipher text) are decrypted by using the value of the IV and the during the encryption process stored secret key (see Lines 15-19).

Listing 4.4: Code Snippet - Decryption Implementation

```
1  ...
2  protected final static int IV_BYTES = 12;
3  ...
4  public byte[] decrypt(byte[] cipherMessage) {
5      ByteBuffer byteBuffer = ByteBuffer.wrap(cipherMessage);
6
7      int ivLength = byteBuffer.getInt();
8      if(ivLength != IV_BYTES) { // check input parameter
9          throw new IllegalArgumentException("invalid iv length");
10     }
11
12     byte[] iv = new byte[ivLength];
13     byteBuffer.get(iv);
14
15     byte[] cipherText = new byte[byteBuffer.remaining()];
16     byteBuffer.get(cipherText);
17
18     final Cipher cipher = getCipher(Cipher.DECRYPT_MODE, iv);
19     return cipher.doFinal(cipherText);
20 }
```

4.5.4 Classification Component

The classification component is used to predict a cost-efficient placement solution for new data objects. To achieve this, we additionally monitor the access information of each data object on class-level. Generally, a data object belongs to a class based on its size. In order to achieve a sufficient class-segmentation, we have decided to divide the data objects into powers of ten depending on the file size.

The implementation of the classification function is shown in Listing 4.5. We simplified the classification process by using the string representation of the data objects' byte size, subtracting one from it and finally using this value for the power of ten (Line 2). To achieve a more fine-granulated class-segmentation, we round the result of the division between the size of the data object in bytes and this calculated value to next power of ten.

For example, if a data object has a size of 60KB (i.e., 60000 Bytes), its string length is therefore 5. By the subtraction of one we then get $10^4 = 10000$. Next, we calculate $60000/10000 = 6$. Because this result is greater than or equal to 5, we use the next power of ten which gives $10 * 10^4 = 100000$. The string representation of this value is then used as the class identifier. Finally, we can say that the data object with the file size of 60KB belongs to the class with identifier 100000.

Listing 4.5: Code Snippet - Classification Implementation

```
1 public static String classify(long bytes) {
2     long classId = (long) Math.pow(10, String.valueOf(bytes).length()
          - 1);
3     if(Math.round((double) bytes / classId) >= 5) {
4         classId = classId * 10;
5     }
6     return String.valueOf(classId);
7 }
```

4.5.5 Update Improvement

Every read or update request performed on a data object fragment is charged by the cloud provider. It often happens that data objects with the same content are updated by the client. Especially, update requests on data object fragments which are stored on long-term storages are very expensive. Furthermore, because of the use of erasure coding, an update request on a data object finally results in n update requests on n different cloud storages. Due to the cost-efficient storage solution, basically, at least $m - n$ data object fragments are stored on long-term storages which makes update request very expensive. Since we also have added two additional components to CORA which actually consume a lot of Central Processing Unit (CPU) power and require long processing times, we have implemented a mechanism which prevents such unnecessary updates.

On each data-altering request (i.e., upload or update) we create a hash which is calculated by taking into account the content length of the data object and the associated access pattern. In case of an upload request, we store the hash value into the database. In case of an update request, we compare the calculated hash value with the hash stored in the database. If both match, we can be sure that the content of the data object stored on the cloud storages corresponds to the content of the update request, so we only need to update the access information by setting the last access time to the time when the update request took place. Otherwise, we must perform a full update for the requested data object. This saves costs and processing time of the entire system.

4.5.6 Global Heuristic Optimization Approach

The most important functionality of the global heuristic optimization approach is the ranking mechanism in order to get a list of the cheapest cloud storages for a particular data object. We calculate the cost of each possible assignment combination of the fragments of a data object to the available cloud storages. For generating the storage ranking, we take the pricing models as basis and generate a cost function based on the different cost components.

The rating mechanism of the classification component decides, how many fragments m of a data object are predicted to be stored on standard cloud storages. The remaining

fragments $n - m$ can be stored on long-term storages. This rating process is shown in Listing 4.6.

As each data object belongs to exactly one class identifier, we can directly derive the associated class from a data object as shown in Line 2. Each class provides four values gathered by the class-monitoring process: the total amount of data object fragments stored on standard (see Line 4) and long-term storages (see Line 5), and the total amount of writes (see Line 14) and reads (see Line 15) on the data object fragments. As shown in Line 12, we first calculate the distribution factor of data object fragments stored on standard storages to the total number of stored data object fragments within the class. Afterwards, we calculate the distribution factor of the amount of reads on the data object fragments to the total access amount of data object fragments in the class (see Line 16). As shown in Line 18, we multiply the result of Line 12, with the result of Line 16 and with the total amount of data object fragments n. Finally, to ensure that maximum m data object fragments are stored on standard storages, we take the minimum value of the rounded result of Line 18 and the value of m.

Listing 4.6: Code Snippet - Classification Implementation

```
1  public static int getOptimizedDataShards(DataObject dataObject) {
2      DataObjectClass doClass = dataObject.getClassification();
3
4      long standardChunkAmount = doClass.getChunksOnStandard()+1;
5      long longTermChunkAmount = doClass.getChunksOnLongTerm()+1;
6
7      //initial time check
8      if(standardChunkAmount == 1 && longTermChunkAmount == 1) {
9          return dataObject.getDataShards();
10     }
11
12     double stdDistFactor = standardChunkAmount / (double) (
           standardChunkAmount + longTermChunkAmount);
13
14     long writes = doClass.getWriteRequests()+1;
15     long reads = doClass.getReadRequests()+1;
16     double readFactor = reads / (double) (writes + reads);
17
18     double dataShards = readFactor * stdDistFactor * dataObject.
           getTotalShards();
19     return Math.min((int) Math.round(dataShards), dataObject.
           getDataShards());
20 }
```

CHAPTER 5

Evaluation

In this chapter, we will describe the evaluation process in detail. First, we will talk about the input track, the digital corpus and the workflow we used for our data creation. Then, we will introduce the selected cloud storages and the general settings of our middleware. Finally, we will explain the functionality of the individual optimization scenarios and their results more precisely.

5.1 Prerequisites

This section describes the starting point for each evaluation scenario. We define the entire process of generating our input data by applying a correlation of an access log file, a digital corpus and a specific method to create the final data set. In addition, we define the used cloud storages, the predefined parameters and the different modes of operation of our middleware, the general system setup for each evaluation process and finally, the baseline that serves as the foundation for evaluating our results.

5.1.1 Input Data

Access Log file

In order to evaluate our global heuristic optimization approach, we use a real-world access log file of a large-scale personal cloud as discussed in [37]. The log file contains about one month, from January to February 2014, of anonymized file access information of about 1.29 million active users and has a file size of about $758GB$. The trace contains the API operations, the session management of the users as well as the file properties such as the file size and a hash that is originated from desktop clients. The trace contains detailed information of about every file access that has happened during the duration of about 30 days.

As the log file holds very much information of data objects which are compressed in a very small period of time, the original log file is inapplicable as input for our evaluation. Therefore, we reduced the log file to a smaller set of data objects. This is done by selecting the used data objects equally among all data objects of the trace in order to include both frequently and seldom used data objects. Further, we reduced the access information by extracting only the relevant properties like file identifier, access time, request type, file size and MIME type.

Since the trace contains information of about data objects that are uploaded prior to the start of the monitoring process of [37], we inserted upload information at the beginning of the log file for all data objects which have read, update or delete request information prior to an upload request. So we can be sure that every request to a data object can be correctly handled by the optimization and does not produce mishits.

Almost all cloud storage providers offer very fine-granulated pricing models with an automated monthly settlement. Most of them offer an additional long-term storage service. Such long-term storages are designed for data that is infrequently used. The storage price for this service is offered at a reduced rate and therefore smaller compared to standard storages. But on the other hand, the request cost are much higher. As our optimization approach searches for a cost-efficient placement solution of data objects, we have to consider the pricing model of long-term storages more precisely. We have to keep in mind that most of the cloud storage providers charge a minimum storage time as well as a minimum file size, especially for long-term storages. The minimum storage duration, often referred to BTU, is 30 days. The minimum file size charged by most of the cloud storage providers is about 1KB. This means, that at the time a 1-byte-sized data object is stored on a long term storage, it will be instantly charged with a storage space of 1KB and a storage duration of 30 days, even if the data object will be immediately deleted or updated. Therefore, when moving a data object from a standard storage to a long term storage, the lower storage price of the latter will be profitable only after a certain days of usage.

Due to the fact that we can only show the correct behavior of our optimization approach if the duration of the log file is significantly greater than the BTU, we have to extend the duration of the access log file mentioned in Section 5.1.1. We do this by repeatedly and randomly selecting a line from the access log file and chronologically appending it to the original log file until we get a resulting trace of about 6 months. During this process, we additionally insert some random access information to prevent a uniform and recurring evaluation output and to further guarantee a realistic access log based on [37]. The insertion of this access information is done by randomly picking a line from the data access logfile, changing the timestamp to the timestamp of the current position in the logfile and setting the request type to “Upload”. This process is randomly executed after every x lines, while x is again a random number in the range from 10 to 50 that turned out as the best choice to gain certain degree of dispersion.

Digital Corpus

To evaluate our extended middleware, as described in Section 4.1.3, against a realistic file set we make use of the digital corpus Govdocs1[1]. Since its creation, the corpus Govdocs1 has been widely used in various research areas. Some examples are papers in the field of file fragment classification [38–40] and information security [41]. Govdocs1 is a very large and heterogeneous corpus created by Digital Corpora[2], which consist of about one million files of various formats collected from the .gov domain by using search engines. The reason for selecting this dataset was to have a wide range of real-world examples instead of generating random files.

Dataset Creation

Finally, we need to implement a function that correlates the access log file with the digital corpus. We developed a method that generates the data set for our evaluation process. For each line of the input trace we read the properties like file identifier, file size and MIME type. Afterwards, we randomly select a file from the corpus which has the same MIME type and also has a file size that is larger than the file size property from the log file. If the method finds a match, it reads in all the bytes from the randomly selected file of the corpus and writes out only the number of bytes which is specified as the file size property in the log file. Finally, we save the new file with the filename of the log file identifier property and the corresponding file extension.

This process results in a realistic data set of approximately 440 files in a file size range from 10 bytes to 51 MB, including various file formats. The total size of our realistic data set is almost 800 MB.

5.1.2 Cloud Storages

Our multi-cloud system architecture uses multiple cloud storages to prevent vendor lock-in and to ensure high availability. Therefore, we have to provide multiple cloud storages to evaluate the functionality of our placement optimization. Since we decided to use a real-world access trace in conjunction with a realistic file set, we also make use of an authentic set of cloud storages in our evaluation. The cloud storage set includes 10 autonomous cloud storages appropriated by 3 different cloud storage providers like Amazon S3, Google Cloud Storage and a self-hosted OpenStack Swift storage. An overview of all used cloud storages is given in Table 5.1.

For each cloud storage provided by Amazon S3[3] and Google Cloud Storage[4], we use the published pricing models accessed on 25^{th} October, 2019. For the self-hosted version based on OpenStack Swift[5], we use the pricing model corresponding to the cloud storage

[1] https://digitalcorpora.org/corpora/files
[2] https://digitalcorpora.org
[3] https://aws.amazon.com/s3/pricing/
[4] https://cloud.google.com/storage/pricing
[5] https://wiki.openstack.org/wiki/Swift

Table 5.1: Used Cloud Storages

	Provider	Region		Type of Storage
1	Amazon	Northern Virginia	US East	Standard
2				IA
3		Northern California	US West	Standard
4		Frankfurt	EU	Standard
5				IA
6		Tokyo	Asia Pacific	Standard
7		Sao Paulo	South America	Standard
8	Google	Europe	Multi-Regional	Standard
9	Self-Hosted	TU Wien	Local	Standard
10				IA

type (e.g., Standard or Infrequent Access (IA)) of Amazon S3 within the region of Frankfurt EU.

5.1.3 Defined Parameters

Since we extended our file access trace to 6 months, we can set the BTU to the correct and real value of 30 days contrary to the work in [10]. We do this due to the fact that almost all cloud storage providers define a minimum storage duration (i.e., BTU) of 30 days. Moreover, to show the correct behavior of our optimization approach, the total duration of the evaluation has to be significantly larger than the value of the BTU. That is one of the reasons why we extended the access logfile to 6 month. As a consequence, we also extended the time step creation interval of CORA to 24 hours, which is twice as long as defined in [9]. We do this because of the extended evaluation duration and due to the fact that [9] holds all the history information in memory. As mentioned in Section 4.1.2, CORA monitors all operations that are performed on a particular data object. Our implemented optimization approach takes into consideration the access history of the 30 most recent time steps (i.e., 30 days, equal to the value of the BTU) to determine a new cost-efficient placement solution. We set this variable to 30 since our preliminary evaluation runs produced the best results for this setup.

Moreover, for each data object $o \in O$, we set the vendor lock-in factor $l_o = 0.5$, the availability $a_o = 99.99\%$ and the durability to $d_o = 99.9999999\%$. Furthermore, we set the erasure code configuration to $EC(2, 3)$. These values are exposed as a very effective configuration setting and proved good results shown in [9]. An overview of the base settings of CORA can be shown in Table 5.2.

5.1.4 Modes of Operation

As we have already described in Section 4.1.3, we implemented the new software components loosely-coupled into the middleware CORA. Each component can therefore be

Table 5.2: Evaluation Settings

Parameter	Variable	Value
Billing Time Unit	BTU	30 d
History Time Used	τ	30 d
Time Step Interval	t_{step}	24 h
vendor lock-in factor	l_o	0.5
availability	a_o	99.99%
durability	d_o	99.9999999%
erasure coding	$EC(m, n)$	$EC(2, 3)$

turned on or off independent from each other. Thus, we can perform a very fine-grained evaluation process to show the improvements of each of our extensions. The possible modes of operation which are provided by our extended middleware SCORA are shown in Table 5.3. In addition, the current effective optimization approach considered by the middleware can be set by three different optimization modes: OPT-No Optimization (NO), OPT-Global Heuristic Optimization (GHO) and OPT-Global Exact Optimization (GEO). The classification component is turned on by using the operational mode CL-ON and turned off with CL-OFF. The same functionality applies to the encryption and the compression components respectively. By default, all additional modes of operation are set to CL-OFF, C-OFF and E-OFF.

5.1.5 Baseline

For a correct evaluation of the functionality of our solution, we have to compare the results to a baseline. In this initial situation, the middleware is set to OPT-NO of mode of operation. The baseline defines the cost that would occur during the evaluation process if we use a fixed set of cloud storages. To show the cost efficiency of our solution, we select a fixed subset of the three cheapest cloud storages mentioned in Section 5.1.2. In our case, these storages are Amazon S3 Northern Virginia Standard, Amazon S3 Frankfurt Standard and the self-hosted OpenStack Swift Standard storage.

5.1.6 System Setup

For our evaluation, we use an Apple iMac(15,1) equipped with a Solid State Disk (SSD) of size 264GB and an Intel Core i7 processor type with a CPU frequency of 4GHz. The processor consists of 4 cores whereas each of them has an L2-Cache of 256KB. The L3-Cache is 8MB and the size of the main memory is 24GB DDR3 (2x4GB + 2x8GB). The used operating system is macOS Mojave in the version 10.14.2.

Table 5.3: Modes of Operation of SCORA

Component	Mode	Functional Description
OPTimizer	OPT-GEO	Each data object which gets monitored by the cloud-based middleware will be placed by the Global Exact Optimization (GEO) approach. See Section 4.3 for more information.
	OPT-GHO	Each data object which gets monitored by the cloud-based middleware will be placed by the Global Heuristic Optimization (GHO) approach. See Section 4.4 for more information.
	OPT-NO	Each data object which gets monitored by the middleware will be placed by a No Optimization (NO) approach, e.g., to a fixed subset of cloud storages defined in Section 5.1.5.
CLassification	CL-ON	Each data object which gets uploaded by the middleware will be classified based on its file size to predict a cost-efficient placement solution of the data object. This component is automatically disabled if no optimization approach (OPT-NO) is selected. See Section 4.1.3 for more information.
	CL-OFF	No classification
Compression	C-ON	Each data object which gets uploaded by the middleware will be compressed by the algorithm gzip. See Section 4.1.3 for more information.
	C-OFF	No compression
Encryption	E-ON	Each data object which gets uploaded by the middleware will be encrypted by the authenticated encryption algorithm AES in the operational mode GCM. See Section 4.1.3 for more information.
	E-OFF	No encryption

5.2 No Optimization Scenario

The first scenario we evaluate is a no optimization scenario where we want to show the improvements of our additionally implemented components compared to the baseline. We use our middleware SCORA and show its cost efficiency without using any placement optimization approach. To achieve this, we disabled the placement optimizer (i.e., Optimizer component) of SCORA just to see the effects of our new components.

For easier readability, we define the no optimization approach with the operational mode C-OFF & E-OFF as NO, and the one with the operational mode C-ON & E-ON as $NO^{C,E}$.

$$\begin{aligned} NO &\widehat{=} \text{ OPT-NO} \\ NO^{C,E} &\widehat{=} \text{ OPT-NO(C-ON, E-ON)} \end{aligned}$$

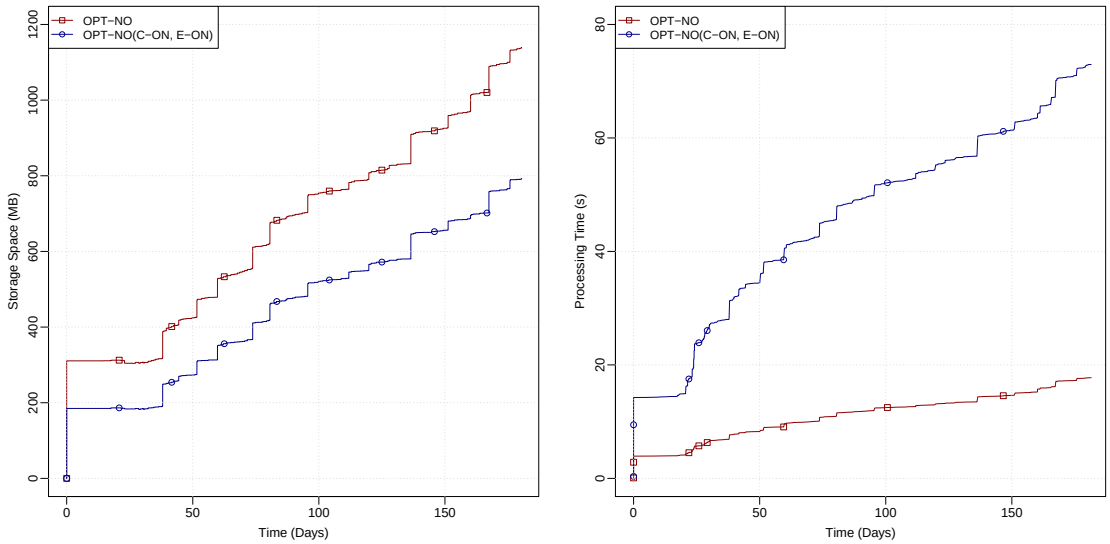
5.2.1 Hypothesis

In this very first step of the evaluation, both NO and $NO^{C,E}$ store new data objects on a fixed subset of cloud storages. The activation of the compression component of $NO^{C,E}$ reduces the required cloud storage space of each data object fragment by using the compression algorithm gzip. This leads to further cost savings since besides the lower storage cost also the transfer size of every read request or migration process is reduced as well. Therefore, the no optimization approach $NO^{C,E}$ with enabled compression component should yield the best results while guaranteeing a secure storage solution. However, due to the time-consuming compression and encryption algorithms, $NO^{C,E}$ will need more processing time than NO.

5.2.2 Execution

The total storage space required on the different cloud storages is shown in Figure 5.1a. Immediately next to it, in Figure 5.1b, shows the total processing time of each executed API request. It should also be noted that any latencies occurred between the client and the cloud storages are not taken into account when measuring the total processing time. The reason for this is that we only want to evaluate the optimization approach and the effects of the different modes of operation of the cloud-based middleware SCORA.

It can be observed from Figure 5.1a that the required cloud storage space at the beginning of the evaluation process rises quickly due to the huge amount of upload requests. Furthermore, it can be seen that the allocated cloud storage space has been reduced for $NO^{C,E}$ to almost half of NO. Right after, it can be observed that the cloud storage space stays nearly consistent which in fact means that no or few upload requests are performed in this period. After about 20 days, it can be seen from the graph that the cloud storage space on both NO and $NO^{C,E}$ lowers a bit due to some delete requests initiated by the client. Further, it can also be seen that deletions on NO have a stronger influence than on $NO^{C,E}$. This is because each data object by using NO is at least greater or equal in



(a) Total Storage Space (b) Total Processing Time

Figure 5.1: Total Storage Space beside the Total Processing Time of the No Optimization Scenario Compared by Applying different Operational Modes

size than using $NO^{C,E}$ and therefore more cloud storage space is freed after a deletion request on NO.

But, the good compression ratio and the high safety achieved by $NO^{C,E}$ results in a deterioration of the overall system's processing time. Compared to Figure 5.1a, it can be seen that in Figure 5.1b the processing time of $NO^{C,E}$ is much higher at initial time compared to NO. The reason for that are the compression and encryption algorithms which involve complex computing processes which require much computing power and memory. Therefore, the total processing time is strongly dependent on the performance of the provided system setup. In particular, the compression and encryption processes are much more time consuming than the decompression and decryption operations. This can be seen especially if we compare the points in time of the increasing cloud storage space of $NO^{C,E}$ in Figure 5.1a with the increasing processing times of $NO^{C,E}$ in Figure 5.1b (e.g., approximately after 40 days or 140 days). Due to the fact that after about 20 days of runtime the total processing time of $NO^{C,E}$ in Figure 5.1b increases and the overall required storage space of $NO^{C,E}$ in Figure 5.1a decreases, we can assume that especially download or update requests are performed at this time, because delete requests have no impact on our measured processing time.

In summary, it can be said that a total of about 30% of the cloud storage space can be saved due to the compression, but about 3.5 times as much processing time is required. Smaller data objects, however, have a major impact on the overall cost, as not only storage cost are decreased, but also any transfer and/or migration cost are reduced.

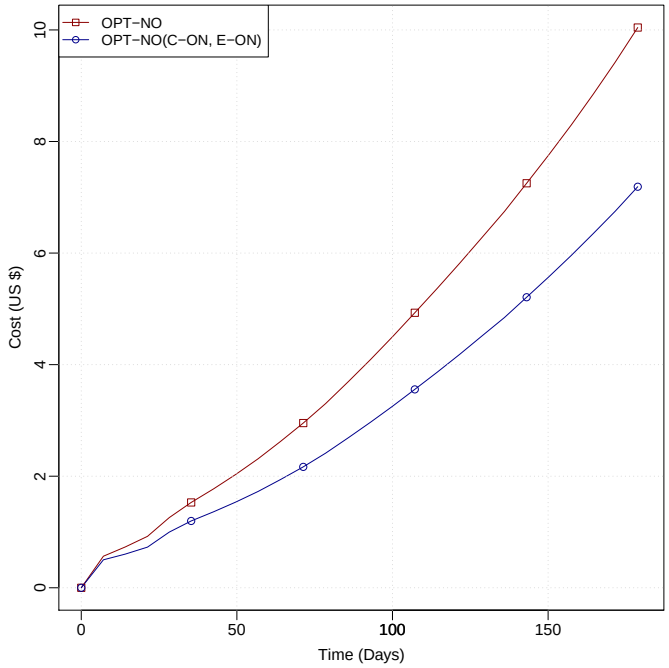


Figure 5.2: Total Cost of the No Optimization Scenario Compared by Applying different Operational Modes

5.2.3 Results

Figure 5.2 shows the cost of the no optimization approach. As before, we compare both the no optimization approach (i.e., the baseline) compared to the no optimization approach with enabled compression and encryption component. In other words, we compare our middleware in OPT-NO mode with OPT-NO(C-ON, E-ON), or in short we check NO against $NO^{C,E}$.

Right at the beginning of the evaluation and after the plenty of upload requests, or more exact after about 7 days, it can be seen from the graph that the cost of NO is higher than $NO^{C,E}$. This is due to the fact that the required storage space is efficiently reduced by the compression component which results in saving cost. As the data object fragments stored on the cloud storages are always compressed if the operational mode C-ON is set, the overall transfer size is much smaller too. Therefore, the longer the evaluation is in progress, the more cost can be saved. At the end of the evaluation, the no optimization approach with enabled compression and encryption $NO^{C,E}$ yields cost savings of about 28%. It can be stated that the cost savings of $NO^{C,E}$ compared to the baseline NO would continuously increase with the duration of the evaluation.

5.3 Global Heuristic Optimization Scenario

The second scenario we are going to evaluate is the global heuristic optimization scenario. We use the middleware SCORA and enable the optimizer component by setting the placement function to the global heuristic optimization approach. To show the cost efficiency, we compare the global heuristic optimization approach to the baseline by applying different modes of operation.

For a clearer understanding and an easier readability, we define the global heuristic optimization approach operated in OPT-GHO as *GHO*, OPT-GHO(C-ON, E-ON) as $GHO^{C,E}$ and OPT-GHO(CL-ON, C-ON, E-ON) as $GHO^{CL,C,E}$. The baseline *NO* is equally defined as in Section 5.2.

NO	$\widehat{=}$	OPT-NO
GHO	$\widehat{=}$	OPT-GHO
$GHO^{C,E}$	$\widehat{=}$	OPT-GHO(C-ON, E-ON)
$GHO^{CL,C,E}$	$\widehat{=}$	OPT-GHO(CL-ON, C-ON, E-ON)

5.3.1 Hypothesis

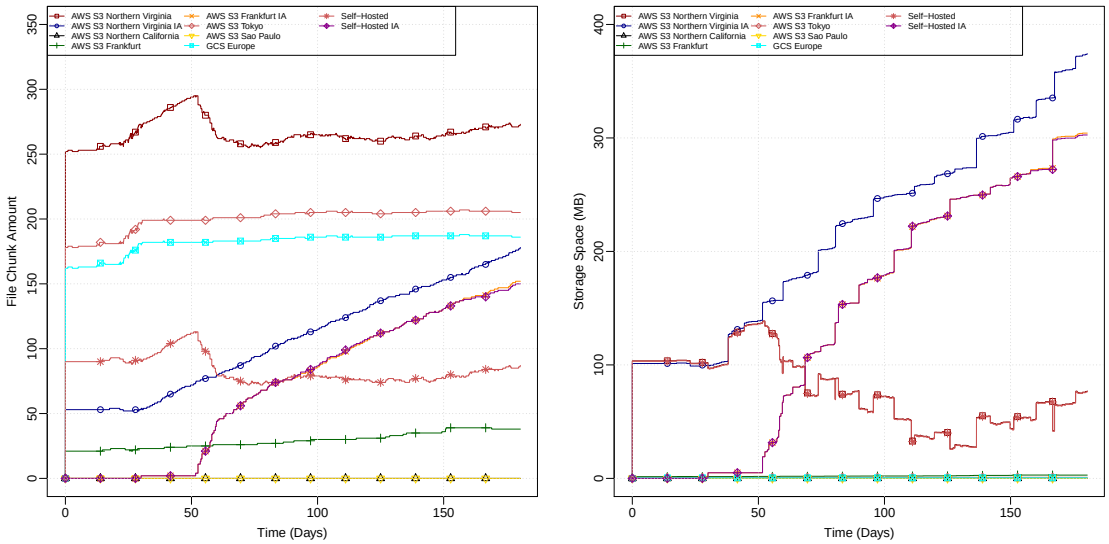
At the very beginning of the evaluation, the optimization approach selects the cheapest set of cloud storages to store new data objects regardless of the applied mode of operation. After at least the time of the BTU, unused data object fragments are expected to be migrated from standard to long-term storages. Furthermore, on every request initiated by the client, the global heuristic optimization approach searches for a cost-efficient placement solution. If a new solution is encountered, rarely or even unused data object fragments are migrated to long-term storages. Moreover, the operational mode C-ON will compress the data objects to minimize the required cloud storage space while the operational mode E-ON will ensure a secure and authenticated storage mechanism. In addition, the mode CL-ON will classify each data object and based on the monitored class-level access information, it will predict a cost-efficient placement solution for the data objects. Therefore, the global heuristic optimization approach with CL-ON & C-ON & E-ON mode of operation should yield the best results and besides guarantees high security, integrity and authenticity.

Finally, the evaluation should give us the following cost ranking of optimization approaches in descending order:

$$NO > GHO > GHO^{C,E} > GHO^{CL,C,E}$$

5.3.2 Execution

Figure 5.3 shows the data object fragment distribution of the global heuristic optimization approach *GHO*. More precisely, Figure 5.3a shows the amount of data object fragments stored on the used cloud storages and Figure 5.3b shows the required cloud storage space.

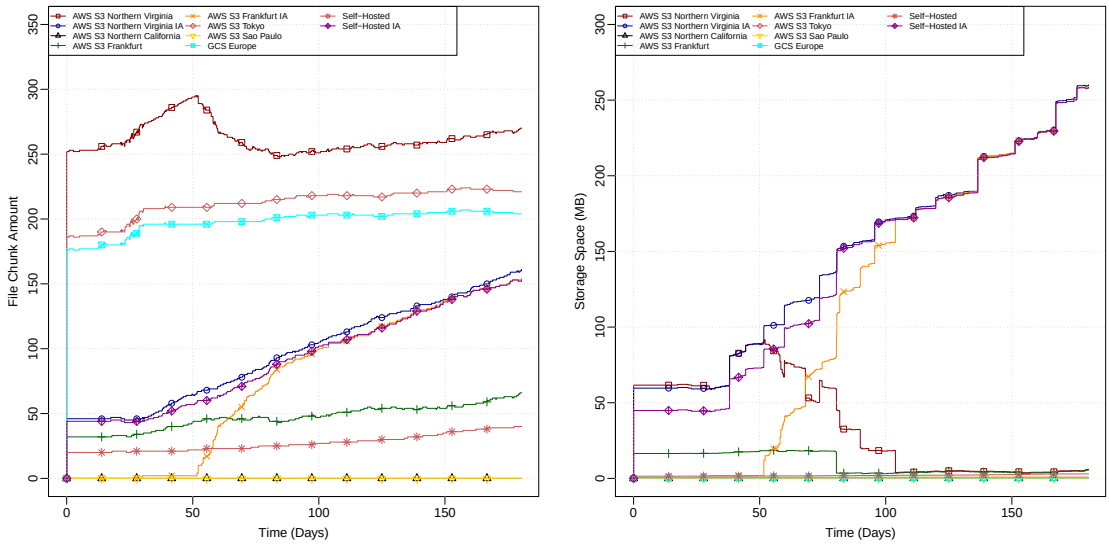


(a) Data Object Fragment Amount (b) Data Object Fragment Storage Space

Figure 5.3: Data Object Fragment Amount beside the Data Object Fragment Storage Space of the Global Heuristic Optimization Scenario in the Operational Mode OPT-GHO

Figure 5.3a shows that right at the beginning of the evaluation most of the data object fragments are stored on the standard storage AWS S3 Northern Virginia because it is the cheapest cloud storage at this point. Most of the other fragments are stored on the standard storages AWS S3 Tokyo and GCS Europe and some of them are distributed to the Self-Hosted standard storage. This is due to the fact that the predefined QoS constraints (i.e, vendor lock-in, availability etc.) must be met by the middleware and these cloud storages are more expensive than AWS S3 Northern Virginia. The third fragment of the rest of the data objects is stored on the long-term storage AWS S3 Northern Virginia IA or the standard storage AWS S3 Frankfurt. The reason why some of the remaining fragments are stored on standard storages is because the optimization approach does not place fragments to long-term storages which are smaller than the BSU of the cloud storage.

Each time a data object is requested, the optimization algorithm calculates the cheapest placement solution of the data object fragments and automatically rearranges the fragment distribution if necessary. It can be observed from the graph that in the period between about 50 days and 60 days, the amount of data object fragments stored on the standard storages AWS S3 Northern Virginia and Self-Hosted decreases, while the amount of fragments on the long-term storages Self-Hosted IA and AWS S3 Frankfurt IA significantly increases. This is because the optimization approach starts migrating unused data object fragments from the standard storages AWS S3 Northern Virginia and Self-Hosted to the long-term storages Self-Hosted IA and AWS S3 Frankfurt IA.



(a) Data Object Fragment Amount

(b) Data Object Fragment Storage Space

Figure 5.4: Data Object Fragment Amount beside the Data Object Fragment Storage Space of the Global Heuristic Optimization Scenario in the Operational Mode OPT-GHO(CL-ON, C-ON, E-ON)

However, Figure 5.3b indicates that the amount of used cloud storage space is considerably higher on long-term storages. This means that in particular larger data object fragments are stored on long-term storages, while the smaller ones are distributed to standard storages. This is because the optimization approach only stores data object fragments on long-term storages whose file size is greater than the BSU of the storage.

At the end of the evaluation, it can be observed from Figure 5.3b that the used standard storage space is only about 10% of the overall used cloud storage space.

Figure 5.4 shows the data object fragment distribution of the global heuristic optimization approach $GHO^{CL,C,E}$, Figure 5.4a shows the amount of data object fragments stored on each cloud storage and Figure 5.4b shows the total used storage space. This means, Figure 5.4 shows the same comparison like Figure 5.3 with the difference that the former uses CL-ON, C-ON and E-ON as mode of operation, which means that the classification, compression and encryption component is enabled during the evaluation process.

The global heuristic optimization approach $GHO^{CL,C,E}$ additionally uses a placement prediction approach to store new data objects based on the history information which is monitored on class-level. Furthermore, the middleware compresses each data object to minimize the total required cloud storages space and besides ensures a secure storage solution achieved by the encryption component.

It can be observed from Figure 5.4a that due to the classification component more data object fragments are initially distributed to the long-term storages AWS S3 Northern

Virginia IA and Self-Hosted IA than in Figure 5.3a. Data object fragments which are predicted to be unused by the classification component are immediately stored on long-term storages. Therefore, a more optimized placement solution can be found earlier for new data objects.

Figure 5.4b shows the corresponding distribution of the storage space of the data object fragments. Compared to Figure 5.3b, it can be observed from Figure 5.4b that at the end of the evaluation almost all of the used cloud storage space is allocated to the long-term storages AWS S3 Northern Virginia IA, AWS S3 Frankfurt IA and the Self-Hosted IA.

5.3.3 Results

Figure 5.5 shows the cost of the global heuristic optimization approach. We compare the cost of the global heuristic optimization approach in different modes of operation to the baseline.

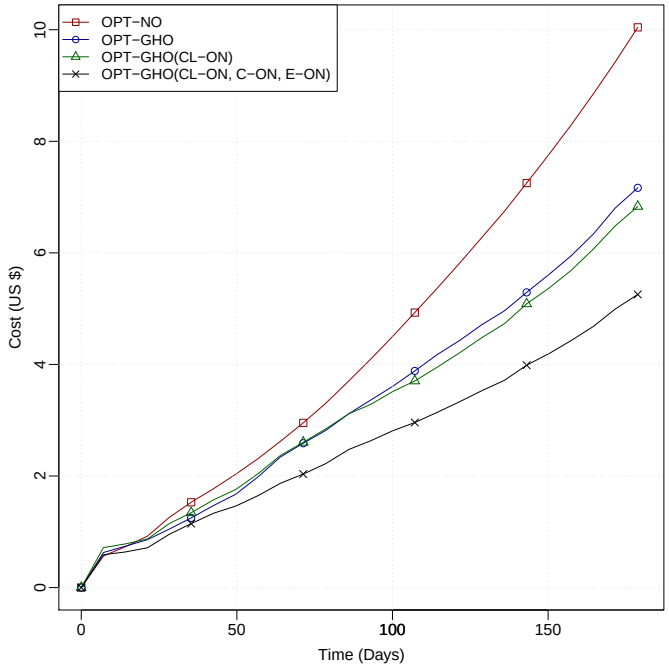


Figure 5.5: Total Cost of the Global Heuristic Optimization Scenario Compared by Applying different Operational Modes

Right at the beginning of the evaluation and after the large number of initial upload requests, it can be seen from the graph that the cost of all optimization approaches are nearly equal. To be more exact, after about 7 days the baseline NO is cheaper than all other optimization approaches. This is because of the BTU of the long-term storages which is instantly billed for each first upload request of a data object fragment. Furthermore, it can be observed from the graph that at this point in time the GHO^{CL}

is the most expensive approach. The reason for this is that the optimization approach GHO^{CL} predicts that more data object fragments should be stored on long-term storages. Between approximately 7 and 25 days runtime of the evaluation the effect of the lower storage price of long-term storages can be seen. After about one month, all optimization approaches GHO, GHO^{CL} and $GHO^{CL,C,E}$ are already cheaper compared to the baseline, while $GHO^{CL,C,E}$ is from this point the cheapest of all. This is due to the fact that the required storage space is significantly reduced by the compression component which in turn minimizes the total cost. Furthermore, as the data object fragments are always stored compressed on the cloud storages if the operational mode C-ON is set, the overall transfer size is much smaller too.

The longer the evaluation is in progress, the more cost can be saved. At the end of the evaluation, the global heuristic optimization approach $GHO^{CL,C,E}$ yields cost savings of about 50% compared to the baseline NO. It can be observed from the graph that the optimization approach $GHO^{CL,C,E}$ increases linearly while the baseline NO rises nearly exponentially. Finally, it can be stated that the cost savings compared to the baseline NO would increase continuously with the duration of the evaluation.

As we already have mentioned, the compression and encryption algorithms require a lot of processing time. Figure 5.6 shows the total processing time of the baseline NO, $NO^{C,E}$, the global heuristic optimization approach GHO and $GHO^{CL,C,E}$, while the latter has all additional components of SCORA enabled.

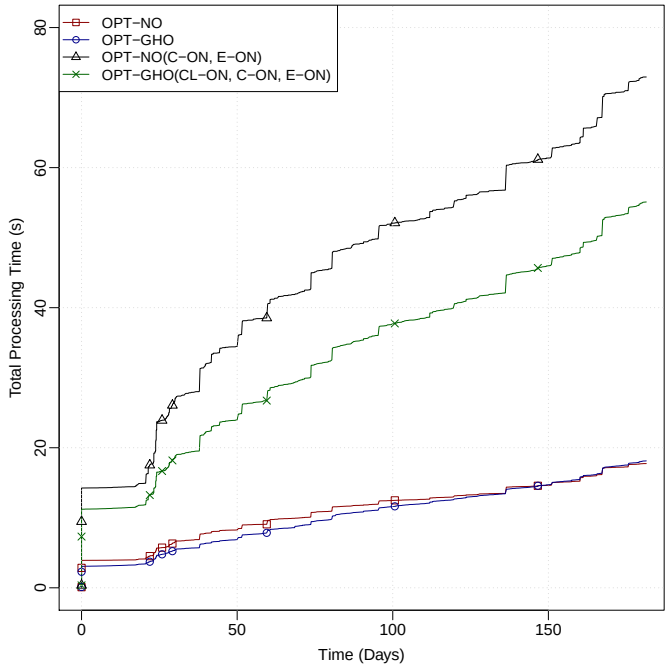


Figure 5.6: Total Processing Time of the Global Heuristic Optimization Scenario Compared by Applying different Operational Modes

It can be observed from Figure 5.6 that right at the beginning GHO needs the lowest processing time. This can be explained by the fact that the global heuristic optimization approach GHO is equipped with an integrated update improvement as explained in Section 4.5.5. As a result, both NO and GHO need nearly the same processing time at the end of the evaluation.

Furthermore, approximately the same behaviour of NO and GHO can be seen when comparing $NO^{C,E}$ and $GHO^{CL,C,E}$ of Figure 5.6. The only difference is that the update improvement affects the total processing time of $GHO^{CL,C,E}$ much more compared to GHO. The heavy computational processings which are required by the compression and encryption component can be skipped if the update improvement detects an update request for a data object fragment which is equal to an already stored one. As a result, at the end of the evaluation the $GHO^{CL,C,E}$ saves about 30% of processing time compared to $NO^{C,E}$.

CHAPTER 6

Conclusion and Future Work

This section summarizes the essential parts of this work. First, we will outline the contributions of this thesis and afterwards we will describe some possibilities in which our work can be extended.

6.1 Contributions

Nowadays, using cloud-based storage services to store data is a popular alternative to traditional local storage systems. Compared to conventional storage architectures, a cloud-based solution can increase data integrity, availability and durability while lowering IT infrastructure cost.

One of the biggest issues of using cloud-based storage systems is the reliance on a particular cloud storage provider. This situation, where the data-owner's needs depend on a single cloud storage provider is denoted as vendor lock-in. Most cloud storage vendors do not ensure profound security guarantees regarding data retention. In addition, storing data on a cloud storage leads to the loss of physical control a data owner has over his data. Therefore, customers have to rely on the security mechanisms and intrusion detection systems the cloud storage vendor provides. Even if a cloud storage vendor might be treated as trustworthy, administration staff or other employees with sufficient permissions can have physical access to sensitive data. These so-called malicious insiders have become a well-known security problem, especially with critical information like medical records or personal data.

In the initial phase of this thesis, we identified the research challenge by analyzing related work in the field of cloud-based storage systems. The research challenge was to design a secure and cost-efficient cloud storage middleware and to implement a global heuristic optimization approach derived from the global exact optimization approach.

To overcome the issues of using cloud-based storage systems, we extended the existing middleware CORA. The result is SCORA, a middleware which uses several independent cloud storage providers to store data objects in a secure, authenticated, redundant and cost-efficient way. This middleware is based on a multi-cloud storage architecture and chooses the cheapest cloud storage provider set while respecting predefined QoS constraints and data access patterns.

The cloud-based middleware optimizes the placement of the data objects in a cost-efficient way. This can be achieved by using a global heuristic optimization approach which fulfills several predefined QoS attributes (e.g., availability, durability and vendor lock-in factor). Moreover, the middleware compresses each data object to reduce the overall storage size which further leads to a reduction of the storage cost. To be invulnerable against possible security breaches by the service provider or a malicious party, our solution ensures that each data object is always transmitted and stored in encrypted form by using an authenticated and strong encryption algorithm. Furthermore, the system monitors the access information of each data object. This historical data is used to find the most suitable cloud storage provider set. The utilization of erasure coding as redundancy mechanism increases the availability and improves the storage efficiency.

After the design and implementation of SCORA, we introduced two evaluation scenarios. In the first scenario, we proved the cost-reducing effect of the additional components provided by SCORA. In the second scenario, we analyzed the cost efficiency of the global heuristic optimization approach where we also outlined the benefits of SCORA. In both scenarios, we showed essential cost savings by comparing the results of each approach to a baseline.

6.2 Future Work

This work provides the design and implementation of a cloud-based middleware which stores data objects in a secure, redundant and cost-efficient way and can therefore be used as foundation for further research projects in the area of cloud-based storage solutions. In the following we show some possible improvements which can be integrated into the existing implementation.

The database of the used storage middleware CORA is currently implemented as an IMDB, which leads to a high memory usage of the overall system. To improve the performance of the system, CORA could migrate to a more powerful solution like using a Structured Query Language (SQL) or No Structured Query Language (NoSQL) database.

Furthermore, CORA is generally implemented as a monolithic application that is based on a centralized system architecture. To increase the performance and scalability of the middleware, CORA could be redesigned to a more modern microservice architecture which distributes intensive data processings (e.g., compression, encryption, erasure coding, etc.) to several independent endpoints.

Additionally, CORA could be extended by an extra caching layer which buffers often used data objects in a local storage system. This can reduce the overall cost enormously because data objects which are cached locally do not have to be requested expensively from the cloud storages.

Finally, the compression component could also be optimized. The classification and monitoring component could track the history of the compression rate of each data object based on its file size and/or MIME type. With this additional historical information, the compression component could apply the most appropriate compression algorithm for each particular data object.

APPENDIX A

Variables

Table A.1: Variables of the Exact Global Optimization Approach

Variable	**Description**
$s \in S = \{s_1, s_2, \ldots\}$	S is the set of available cloud storages, where s is a particular cloud storage.
$o \in O = \{o_1, o_2, \ldots\}$	O is the set of all data objects, where o is a particular data object.
$f \in F_o = \{f_{o1}, f_{o2}, \ldots\}$	F_o is the set of all data object fragments, where f is a particular data object fragment.
n	n is the amount of data object fragments, whereby each data object o is split into n fragments (i.e., $\lvert F_o \rvert = n$).
m	m is the amount of data object fragments that are required to successfully reconstruct a data object o.
$EC(m, n)$	$EC(m, n)$ is the erasure code configuration with the mentioned parameters m and n.
$b_s \in B_s^{st} = \{b_{s1}, b_{s2}, \ldots\}$	B_s^{st} is the set of all price steps of the storage cost for a storage s, where b_s defines a particular price step.
$b_s \in B_s^{T_{out}} = \{b_{s1}, b_{s2}, \ldots\}$	$B_s^{T_{out}}$ is the set of all price steps of outgoing data transfer cost of a storage s, where b_s defines a particular price step.
	Continued on next page

Variable	Description
$b_s = (b_s^L, b_s^U, p_s)$	b_s^L is the lower bound of the price step b_s, b_s^U is the upper bound of the price step b_s and p_s is the actual price within the given range of a cloud storage s.
$c_{(s,f,\tau)}$	$c_{(s,f,\tau)}$ are the total cost for storing a data object fragment f on a cloud storage s, while taking into account the history information of the last τ minutes.
$c^S_{(s,f,\tau)}$	$c^S_{(s,f,\tau)}$ are the storage cost.
$c^R_{(s,f,\tau)}$	$c^R_{(s,f,\tau)}$ are the read request cost.
$c^W_{(s,f,\tau)}$	$c^W_{(s,f,\tau)}$ are the write request cost.
$c^{T_{in}}_{(s,f,\tau)}$	$c^{T_{in}}_{(s,f,\tau)}$ are the incoming data transfer cost.
$c^{T_{out}}_{(s,f,\tau)}$	$c^{T_{out}}_{(s,f,\tau)}$ are the outgoing data transfer cost.
$c^M_{(s_1,s_2,f)}$	$c^M_{(s_1,s_2,f)}$ are the migration cost for transferring a data object fragment f from a cloud storage s_1 to a cloud storage s_2, where s_1 and s_2 are appropriated by different cloud storage providers.
$c^{M_{red}}_{(s_1,s_2,f)}$	$c^{M_{red}}_{(s_1,s_2,f)}$ are the migration cost for transferring a data object fragment f from a cloud storage s_1 to a cloud storage s_2, where s_1 and s_2 are appropriated by the same cloud storage provider.
$\beta_{(s,f)}$	$\beta_{(s,f)}$ is the number of transferred bytes of a cloud storage s in the current billing period.
$\gamma_{(s,f)}$	$\gamma_{(s,f)}$ is the number of used storage space of a cloud storage s in the current billing period.
$\sigma_{(f,\tau)}$	$\sigma(f,\tau)$ is the size of a data object fragment f in the last τ minutes.
$\hat{\sigma}_{(f,BTU)}$	$\hat{\sigma}_{(f,BTU)}$ is the size of a data object fragment f that is charged for the remaining time of the BTU.
$p^S_{(s,\gamma_{(s,f)})}$	$p^S_{(s,\gamma_{(s,f)})}$ is the storage price of a data object fragment f and a cloud storage s.
p^R_s	p^R_s is the storage price of a data object fragment f and a cloud storage s.

Continued on next page

Variable	Description
$p^{S}_{(s,\gamma_{(s,f)})}$	$p^{S}_{(s,\gamma_{(s,f)})}$ is the storage price of a data object fragment f and a cloud storage s.
p^{R}_{s}	p^{R}_{s} is the read request price of a data object fragment f and a cloud storage s.
p^{W}_{s}	p^{W}_{s} is the write request price of a data object fragment f and a cloud storage s.
$p^{T_{in}}_{(s,\beta_{(s,f)})}$	$p^{T_{in}}_{(s,\beta_{(s,f)})}$ is the incoming data transfer price of a data object fragment f and a cloud storage s.
$p^{T_{out}}_{(s,\beta_{(s,f)})}$	$p^{T_{out}}_{(s,\beta_{(s,f)})}$ is the outgoing data transfer price of a data object fragment f and a cloud storage s.
p^{ret}_{s}	p^{ret}_{s} is the retrieval price of a data object f and a cloud storage s.
$r^{R}_{(f,\tau)}$	$r^{R}_{(f,\tau)}$ is the number of read requests performed on a data object fragment f in the last τ minutes.
$r^{W}_{(f,\tau)}$	$r^{W}_{(f,\tau)}$ is the number of write requests performed on a data object fragment f in the last τ minutes.
$t^{in}_{(f,\tau)}$	$t^{in}_{(f,\tau)}$ defines the number of bytes written to a cloud storage s in the last τ minutes.
$t^{out}_{(f,\tau)}$	$t^{out}_{(f,\tau)}$ defines the number of bytes read to a cloud storage s in the last τ minutes.
$l^{cl}_{(s,f)}$	$l^{cl}_{(s,f)}$ defines the probability value on class-level that a data object fragment f will be stored on a standard storage s.
$l_{(s,f)}$	$l_{(s,f)}$ defines the penalty factor of a data object fragment f and a cloud storage s.

Table A.2: Decision Variables of the Exact Global Optimization Approach

Variable	**Description**
$x_{(s,f)} \in \{0,1\}$	If a data object fragment f is stored on a cloud storage s, then $x_{(s,f)} = 1$, otherwise $x_{(s,f)} = 0$.
$h_f \in \{0,1\}$	If a data object fragment f is stored on a long-term cloud storage s, then $h_f = 1$, otherwise $h_f = 0$.
$\hat{h}_f \in \{0,1\}$	If a cloud storage s is a long-term cloud storage, then $\hat{h}_f = 1$, otherwise $\hat{h}_f = 0$.
$z_{(s_1,s_2)} \in \{0,1\}$	If two cloud storages s_1 and s_2 are provided by different cloud storage provider, then $z_{(s_1,s_2)} = 1$, otherwise $z_{(s_1,s_2)} = 0$.
$y_{(s_1,s_2)} \in \{0,1\}$	If two cloud storages s_1 and s_2 are provided by the same cloud storage provider, then $y_{(s_1,s_2)} = 1$, otherwise $y_{(s_1,s_2)} = 0$.
$g_{(\tilde{S},f)} \in \{0,1\}$	If every cloud storages $s \in \tilde{S}$ has stored a data object fragment $f \in F_o$, then $g_{(\tilde{S},f)} = 1$, otherwise $g_{(\tilde{S},f)} = 0$.
$u^{st}_{(s,b_s)} \in \{0,1\}$	If the used storage space of a cloud storage s is greater than the lower bound b^L_s, then $u^{st}_{(s,b_s)} = 1$, otherwise $u^{st}_{(s,b_s)} = 0$.
$v^{st}_{(s,b_s)} \in \{0,1\}$	If the used storage space of a cloud storage s is lower than the upper bound b^U_s, then $v^{st}_{(s,b_s)} = 1$, otherwise $v^{st}_{(s,b_s)} = 0$.
$o^{st}_{(s,b_s)} \in \{0,1\}$	If the used storage space of a cloud storage s is between the lower bound b^L_s and the upper bound b^U_s, then $o^{st}_{(s,b_s)} = 1$, otherwise $o^{st}_{(s,b_s)} = 0$.
$u^{T_{out}}_{(s,b_s)} \in \{0,1\}$	If the outgoing bytes of data transfer of a cloud storage s is greater than the lower bound b^L_s, then $u^{T_{out}}_{(s,b_s)} = 1$, otherwise $u^{T_{out}}_{(s,b_s)} = 0$.
$v^{T_{out}}_{(s,b_s)} \in \{0,1\}$	If the outgoing bytes of data transfer of a cloud storage s is lower than the upper bound b^U_s, then $v^{T_{out}}_{(s,b_s)} = 1$, otherwise $v^{T_{out}}_{(s,b_s)} = 0$.
$o^{T_{out}}_{(s,b_s)} \in \{0,1\}$	If the outgoing bytes of data transfer of a cloud storage s is between the lower bound b^L_s and the upper bound b^U_s, then $o^{T_{out}}_{(s,b_s)} = 1$, otherwise $o^{T_{out}}_{(s,b_s)} = 0$.

APPENDIX B

Parameters

Table B.1: Predefined Parameters of the Additional Components

Component	**Parameter**	**Value**
Compression	Algorithm	gzip
Encryption	Algorithm	AES
	Padding	No Padding
	Mode of Operation	GCM
	Key Size	128 Bit
	Initialization Vector Size	96 Bit
	Authentication Tag Size	128 Bit
Classification	Class Segmentation	File Size Based

List of Figures

List of Tables

List of Algorithms

Acronyms

AD Associated Data. 17, 18

AE Authenticated Encryption. 17, 18

AEAD Authenticated Encryption with Associated Data. 17, 18, 32, 50

AES Advanced Encryption Standard. 16, 18, 24, 32, 50, 51, 62, 81

API Application Programming Interface. 1, 22, 27–30, 49, 57, 63

BRP Block Rate Pricing. 21–26, 33

BSU Billing Storage Unit. 34, 45, 67, 68

BTU Billing Time Unit. 34, 45, 48, 49, 58, 60, 61, 66, 69, 78

CBC-MAC Cipher Block Chaining Message Authentication Code. 17

CIA Confidentiality, Integrity, Availability. 14, 16–18

CORA COst-efficient data RedundAncy in the cloud. 3, 4, 27–31, 33, 42, 49, 50, 54, 60, 74, 75

CPU Central Processing Unit. 54, 61

CRUD Create Read Update Delete. 27–29

CTR Counter Mode. 50

DES Data Encryption Standard. 15, 16

DoS Distributed Denial of Service. 14

DoS Denial of Service. 14

EDE Encrypt-Decrypt-Encrypt. 15

GCM Galois/Counter Mode. 17, 18, 32, 50–52, 62, 81

GEO Global Exact Optimization. 61, 62

GHO Global Heuristic Optimization. 61, 62, 66

GMAC Galois Message Authentication Code. 18

gzip GNU zip. 20, 32, 62, 63, 81

HMAC Hash-based Message Authentication Code. 17

IA Infrequent Access. 60

IaaS Infrastructure as a Service. 9

IMDB In-Memory Database. 30, 32, 74

IT Information Technology. xi, 1, 10, 73

IV Initialization Vector. 51, 52

JCE Java Cryptography Extension. 50

MAC Message Authentication Code. 17, 18, 26, 50

MILP Mixed Integer Linear Programming. 21, 22

MIME Multipurpose Internet Mail Extensions. 4, 5, 22, 31, 58, 59, 75

NIST National Institute of Standards and Technology. 8, 9, 16, 51

NO No Optimization. 61–63, 66

NoSQL No Structured Query Language. 74

NP-Problem Non deterministic Polynomial Problem. 13

OMAC One-key Message Authentication Code. 17

P-Problem Polynomial Problem. 13

P2P Peer-to-Peer. 3

PaaS Platform as a Service. 9

PMAC Parallelizable Message Authentication Code. 17

QoS Quality-of-Service. xi, 2, 3, 5, 7, 21, 37, 41, 67, 74

RAID Redundant Array of Independent Disks. 19, 21, 24, 29

REST Representational State Transfer. 49

SaaS Software as a Service. 9, 10

SCORA Secure, COst-efficient data RedundAncy in the cloud. 26, 27, 31, 44, 61–63, 66, 70, 74

SHA Secure Hash Algorithm. 24

SLA Service Level Agreement. 10

SLO Service Level Objective. 30, 41

SQL Structured Query Language. 74

SSD Solid State Disk. 61

SSL Secure Sockets Layer. 18

StaaS Storage as a Service. 10

TDEA Triple Data Encryption Algorithm. 15

TDES Triple Data Encryption Standard. 15

TLS Transport Layer Security. 18

UMAC Universal-hashed Message Authentication Code. 17

UX User Experience. 19

WWW World Wide Web. 7

Bibliography

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the Association for Computing Machinery*, vol. 53, no. 4, pp. 50–58, 2010.

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

[3] E. Allen and C. M. Morris, "Library of congress and duracloud launch pilot program using cloud technologies to test perpetual access to digital content," 2009, accessed: October 2019. [Online]. Available: https://www.loc.gov/item/prn-09-140/library-of-congress-and-duracloud-launch-pilot-program/2009-07-14/

[4] B. Butler, "Gartner: Top 10 cloud storage providers," *Network World*, 2013, accessed: October 2019. [Online]. Available: https://www.networkworld.com/article/2162466/cloud-computing-gartner-top-10-cloud-storage-providers.html

[5] B. Satzger, W. Hummer, C. Inzinger, P. Leitner, and S. Dustdar, "Winds of change: From vendor lock-in to the meta cloud," *IEEE Internet Computing*, no. 1, pp. 69–73, 2013.

[6] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, "Racs: A case for cloud storage diversity," in *Proceedings of the 1st ACM Symposium on Cloud Computing*. ACM, 2010, pp. 229–240.

[7] D. Bermbach, T. Kurze, and S. Tai, "Cloud federation: Effects of federated compute resources on quality of service and cost," in *IEEE International Conference on Cloud Engineering*, 2013, pp. 31–37.

[8] M. Schnjakin, T. Metzke, and C. Meinel, "Applying erasure codes for fault tolerance in cloud-raid," in *IEEE 16th International Conference on Computational Science and Engineering*, 2013, pp. 66–75.

[9] P. Waibel, C. Hochreiner, and S. Schulte, "Cost-efficient data redundancy in the cloud," in *IEEE 9th International Conference on Service-Oriented Computing and Applications*, 2016, pp. 1–9.

[10] 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.

[11] P. Kumari and P. Kaur, "A survey of fault tolerance in cloud computing," *Journal of King Saud University - Computer and Information Sciences*, 2018.

[12] M. Alhamad, T. Dillon, and E. Chang, "Conceptual SLA framework for cloud computing," in *IEEE 4th International Conference on Digital Ecosystmes and Technologies*, 2010, pp. 606–610.

[13] J. Wu, L. Ping, X. Ge, Y. Wang, and J. Fu, "Cloud storage as the infrastructure of cloud computing," in *2010 International Conference on Intelligent Computing and Cognitive Informatics*, 2010, pp. 380–383.

[14] B. Selby, "What is cloud storage and what are its advantages?" *Cloud Storage Advice*, 2018, accessed: October 2019. [Online]. Available: https://cloudstorageadvice.com/what-is-cloud-storage/

[15] S. Obrutsky, "Cloud storage: Advantages, disadvantages and enterprise solutions for business," in *Cloud Storage: Advantages, Disadvantages and Enterprise Solutions for Business*, 2016.

[16] A. Pritchett, "6 pros and cons of cloud storage for business," *Compare the Cloud*, 2018, accessed: October 2019. [Online]. Available: https://www.comparethecloud.net/articles/6-pros-and-cons-of-cloud-storage-for-business/

[17] "Pros and cons of cloud storage," *Secure Storage Services*, accessed: October 2019. [Online]. Available: https://www.securestorageservices.co.uk/article/11/pros-and-cons-of-cloud-storage

[18] M. Schnjakin and C. Meinel, "Evaluation of cloud-raid: A secure and reliable storage above the clouds," in *22nd International Conference on Computer Communication and Networks*, 2013, pp. 1–9.

[19] F. Rothlauf, *Design of Modern Heuristics: Principles and Application*, 1st ed. Springer Publishing Company, Incorporated, 2011.

[20] S. Karen, J. Wayne, and T. Miles, "Confidentiality, integrity, and availability," *MDN Web Docs*, 2008, accessed: October 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Information_Security_Basics/Confidentiality,_Integrity,_and_Availability

[21] G. Singh and S. Kinger, "A study of encryption algorithms (rsa, des, 3des and aes) for information security," *International Journal of Computer Applications*, vol. 67, pp. 33–38, 2013.

[22] M. Bellare, R. Canetti, and H. Krawczyk, "Message authentication using hash functions— the hmac construction," vol. 2, no. 1, 1996.

[23] D. McGrew, "Efficient authentication of large, dynamic data sets using galois/counter mode (gcm)," in *3rd International IEEE Security in Storage Workshop*. IEEE, 2005, pp. 89–94.

[24] B. Yang, S. Mishra, and R. Karri, "High speed architecture for galois/counter mode of operation (gcm)," *IACR Cryptology ePrint Archive*, vol. 2005, p. 146, 2005.

[25] 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 Transactions on Cloud Computing*, vol. 3, no. 3, pp. 372–386, 2015.

[26] W. Wang, P. Li, L. Han, S. Huang, K. Xu, C. Yu, and J. Lei, "An enhanced erasure code-based security mechanism for cloud storage," *Mathematical Problems in Engineering*, vol. 2014, pp. 1–8, 2014.

[27] H. Weatherspoon and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 328–338.

[28] F. Tsapeli and V. Tsaoussidis, "Routing for opportunistic networks based on probabilistic erasure coding," in *Wired/Wireless Internet Communication*, Y. Koucheryavy, L. Mamatas, I. Matta, and V. Tsaoussidis, Eds. Springer Berlin Heidelberg, 2012, pp. 257–268.

[29] L. Čegan, "Empirical study on effects of compression algorithms in web environment," *Journal of Human Capital Development*, 2017.

[30] V. Krasnov, "Results of experimenting with brotli for dynamic web content," *The Cloudflare Blog*, 2018, accessed: October 2019. [Online]. Available: https://blog.cloudflare.com/results-experimenting-brotli/

[31] T. G. Papaioannou, N. Bonvin, and K. Aberer, "Scalia: An adaptive scheme for efficient multi-cloud storage," *International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–10, 2012.

[32] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 292–308.

[33] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "Depsky: Dependable and secure storage in a cloud-of-clouds," *ACM Transactions on Storage*, vol. 9, no. 4, pp. 12:1–12:33, 2013.

[34] K. D. Bowers, A. Juels, and A. Oprea, "Hail: A high-availability and integrity layer for cloud storage," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM, 2009, pp. 187–198.

[35] J. Raigoza and K. Jituri, "Evaluating performance of symmetric encryption algorithms," in *International Conference on Computational Science and Computational Intelligence*, 2016, pp. 1378–1379.

[36] P. Favre-Bulle and P. Favre-Bulle, "Security best practices: Symmetric encryption with aes in java and android," *ProAndroidDev*, 2018, accessed: October 2019. [Online]. Available: https://proandroiddev.com/security-best-practices-symmetric-encryption-with-aes-in-java-7616beaaade9

[37] R. G. Tinedo, Y. Tian, J. Sampé, H. Harkous, J. Lenton, P. G. López, M. S. Artigas, and M. Vukolic, "Dissecting ubuntuone: Autopsy of a global-scale personal cloud back-end," in *Internet Measurement Conference*, 2015.

[38] L. Hiester and J. Jarman, "File fragment classification using neural networks with lossless representations," in *File Fragment Classification Using Neural Networks with Lossless Representations*, 2018.

[39] K. Beneduce, "Attributes and machine learning for fragment identification and malware analysis." Monterey, California: Naval Postgraduate School, 2014.

[40] J. Young, K. Foster, S. Garfinkel, and K. Fairbanks, "Distinct sector hashes for target file detection," *Computer*, vol. 45, no. 12, pp. 28–35, 2012.

[41] N. Scaife, H. Carter, P. Traynor, and K. R. B. Butler, "Cryptolock (and drop it): Stopping ransomware attacks on user data," in *IEEE 36th International Conference on Distributed Computing Systems*, 2016, pp. 303–312.