Cost-Efficient Data Redundancy in the Cloud

Philipp Waibel, Christoph Hochreiner, Stefan Schulte
Distributed Systems Group, TU Wien, Austria
Email: {p.waibel, c.hochreiner, s.schulte}@infosys.tuwien.ac.at

***Abstract*—Nowadays, the usage of cloud storages to store data is a popular alternative to traditional local storage systems. However, besides the benefits such services can offer, there are also some downsides like vendor lock-in or unavailability. Furthermore, the large number of available providers and their different pricing models can turn the search for the best fitting provider into a tedious and cumbersome task. Furthermore, the optimal selection of a provider may change over time.**

In this paper, we formalize a system model that uses several cloud storages to offer a redundant storage for data. The according optimization problem considers historic data access patterns and predefined Quality of Service requirements for the selection of the best-fitting storages. Through extensive evaluations we show the benefits of our work and compare the novel approach against a baseline which follows a state-of-the-art approach.

***Index Terms*—Cloud storage; Redundant storage; Erasure coding; Vendor lock-in; Long-term storage; Fault tolerance**

I. Introduction

The usage of cloud storages is a popular way to store data in an accessible and reliable way. Companies, government organizations and even private persons use cloud storages as an alternative to maintaining their own storage systems [1]. Apart from increasing the availability and durability of data, these services can lower the cost of the storage for the customer. While a private storage system may be feasible and economical for big enterprises, it is in many cases not cost-efficient for small and medium-sized enterprises [2]. Cloud storage systems can lead to cost reductions due to the decrease in IT maintenance cost that would occur for a private storage.

Nowadays, several publicly available cloud storage providers exists, e.g., Amazon S3[1], Google Cloud Storage[2], or RackSpace CloudFiles[3] [3]. These providers offer the customer easy to use Web and API interfaces that hide the complexity of the storage systems.

The decision where the data should be stored is not trivial. To choose the best-fitting provider, a customer has to take several constraints into account, e.g., which providers to avoid, which geographical locations and pricing models of potential providers to select, or the offered storage technology. Also, cost should usually be minimized. Pricing models vary significantly depending on the data amount to be stored and the individual data access patterns. Also, apart from "standard" storages, specialized long-term storage services like Amazon Glacier[4] exist, which offer lower cost but also decreased Quality of Service (QoS), e.g., data retrieval may take hours.

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

Relying on only one cloud provider may lead to additional issues: A provider could, e.g., increase the price of the storage or go out of business [4]–[6]. This can result in the need to migrate the data to another provider, which involves migration cost and implementation or administration efforts. If a provider goes out of business, the data may even be lost, if there is no further replica of the data available. In addition, even big cloud storage providers struggle with service outages [7]. Moreover, cloud providers' terms of usage and customer properties may evolve over the time, e.g., because cloud providers modify their pricing models or simply the amount of data a customer stores changes. To avoid these issues, the redundant use of several providers is necessary. Besides decreasing the level of vendor lock-in [8], the usage of different providers increases the durability and availability of the data.

Within the work at hand, we address the problem of cost-efficient data redundancy in the cloud. We do this by formulating an optimization problem that optimizes the placement of files, in the remainder of this paper called data objects, on several cloud storage providers in a redundant and cost-efficient way. Furthermore, the optimization ensures that predefined storage requirements (i.e., availability, durability and vendor lock-in factor) of the customer are satisfied at any time. The optimization takes also the access pattern for the data objects, which is a significant cost factor [9], into account. We, further, present a cloud-based storage middleware, called *CORA*, that uses Mixed Integer Linear Programming (MILP) to solve the optimization problem. The middleware continuously monitors the usage of the stored data objects and dynamically rearranges the placement, with the help of the optimization, if a new cost-efficient storage solution can be realized.

The remainder of this paper is organized as follows: In Section II, we provide background information for our approach. Subsequently, we describe CORA in Section III. Afterwards, we present the linear optimization problem. The evaluation setup is described in Section V and the results of the evaluation are described in Section VI. Section VII discusses the related work. Section VIII concludes the paper and provides an outlook on our future work.

II. Background

Before we discuss the data object placement approach, we need to define some preliminaries.

A. Quality of Service

Several QoS aspects need to be considered when storing data in the cloud. These include availability, durability, and

the vendor lock-in factor, which will be regarded within our optimization approach as follows:

1) Availability: Defines the probability that a service (here: a storage service), is up and running for a specific time span [10]. This parameter is declared as the availability of the service over a given time span in percent, e.g., 99.99% over a year.

2) Durability: Defines the probability that data in a storage service does not get lost, e.g., due to a hardware failure. This parameter is declared as the durability of stored objects in percent over a given time span, e.g., 99.999999% over a year.

3) Vendor lock-in factor: Defines the situation when data is locked on one provider and, thus, cannot be migrated to another one [9]. For example, if the service provider is not accessible for some time or, in the worst case, goes out of business [7], [8]. This parameter is declared as a value between $(0, 1]$ and is calculated by $lockin = \frac{1}{N}$ where N is the number of providers that are used to store a file.

Usually, consumers define the required QoS attributes in Service-Level Objectives (SLOs) [10]. The optimization and CORA are designed to meet the customer-defined SLOs while minimizing the overall storage cost.

B. Erasure Coding

Erasure Coding is a redundancy mechanism where a data object is split into n data object chunks in a way that the whole data object can be reconstructed by any subset of size m ($m < n$) of those data object chunks [11]–[13]. Erasure coding is defined by the tuple (m, n). With this characteristic, erasure coding is a superset of a RAID system but also of a normal replication system [14]. For example, a RAID 5 can be described by setting $m = 4$ and $n = 5$. Furthermore, a normal replication can be achieved by setting $m = 1$ and $n = 3$ which will generate three replications. The main advantage of using erasure coding instead of replication is the smaller additional storage needed to achieve the same level of redundancy [14].

C. Pricing Models

Pricing models vary between different cloud storage providers. Nevertheless, most of the models are based on a similar notion: The pricing models account for the used storage, the used outgoing transfer and the number of read and write requests. Notably, most providers do not charge anything for incoming transfer as well as for deleting data. Further, most providers reduce the price for a higher usage of the system, e.g., the more storage space is leased, the cheaper it is to store additional data. This is also known as a *Block Rate Pricing model* [15]. If data has to be migrated from one provider to another, additional migration cost may be charged. In general, for migration the outgoing transfer and the incoming transfer are charged. However, especially large providers have often several geographical distributed data centers, called regions, and charge a reduced price to migrate data among the regions.

Beside conventional storages, some providers also offer long-term storages. Those storage solutions are optimized for longer storage with none or rare access and have therefore a reduced storage price but a higher access price, which may also include data retrieval cost. Additionally, long-term storage solutions often have a minimum storage duration, i.e., a *Billing Time Unit* (BTU). After storing a file on such a long-term storage, charging is done for the complete BTU, independent of the fact that the storage may no longer be used. For some storage systems there is also a minimum object size that defines the minimum size that is billed, i.e., a *Billing Storage Unit* (BSU). Smaller data objects will be charged for the complete BSU, despite the fact that only part of the storage size is actually utilized.

Besides several geographically distributed regions several cloud storage providers also offer different storage technologies. For example, Amazon offers following storage solutions: Standard Storage, Reduced Redundancy Storage, Standard-Infrequent Access (IA) Storage, and Glacier Storage. Each storage option has different properties and prices, e.g., the Standard-IA storage has a cheaper storage price, but also a smaller availability in comparison to the standard storage. Further, the Standard-IA storage has a BTU of 30 days and a BSU of 128 KB.

While the pricing models of most large cloud storage providers are based on a similar notion there are some differences between them. For instance, while Google Cloud Storage also offers a price reduction for the outgoing data transfer, they do not apply a Block Rate Pricing model for the used storage.

TABLE I
SLO ATTRIBUTE EXAMPLE

File Name	Availability (%)	Durability (%)	Vendor lock-in
image.png	99.8	99.999	0.4
backup.tar	99.9	99.99999	0.5

III. CORA

Before we formulate the optimization problem we describe the middleware that uses MILP to solve the problem. *CORA* is a middleware with the following core features:

- CORA stores data objects on different cloud storages in a redundant way by applying erasure coding.
- CORA optimizes the placement of data objects on all storages in a cost-efficient way without violating user-defined SLOs.
- CORA uses long-term storages to store not or rarely used data objects.
- If CORA recognizes the unavailability of a storage, it recreates the missing data object chunks and uploads them to alternative storages to restore data redundancy.

With these core features, *CORA* provides a cost-efficient solution to store data with a high availability, while not relying on only one specific storage provider.

As mentioned before, the predefined SLOs that are taken into account for the data object placement selection are availability, durability and vendor lock-in factor. These SLOs are defined for each data object by the data owner. Table I presents

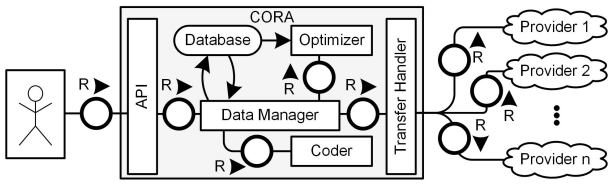


Fig. 1. CORA – Architecture

two example files with defined SLOs. For example, the file with the name *image.png* has to be stored in a way that it has an availability of 99.8% over a year and a durability of 99.999% over a year. Further, it has to be stored in a way that the file can be recreated by using the data from at most three storages, due to the defined lock-in value of 0.4. While the SLOs availability and durability are used for the selection of the particular storages, the vendor lock-in factor is used to select the required erasure code configuration values.

Figure 1 depicts the general architecture of CORA using FMC notation. The central component of the middleware is the *Data Manager*. It is responsible for the coordination between the components of CORA and, thereby, for the read and write process of data objects from and to the cloud storages.

To be able to store the data objects in a redundant way, we apply erasure coding. The component *Coder* is responsible for this redundancy mechanism. It transforms the data according to the erasure coding parameters n and m, as discussed in Section II-B. The output of the Coder is, for a storage process, the data object transformed into multiple data object chunks and, for the read process, a single data object that was created from several data object chunks. To decrease the read cost, CORA does not read all data object chunks, instead it only reads enough data object chunks to restore a data object. Furthermore, the component is responsible for the recreation of a data object chunk if a data object chunk is damaged or the storage is not available. The recreation is done by using the remaining available data object chunks.

The *Optimizer* component is responsible for optimizing the placement of data objects in a cost-efficient way, taking into account the user-defined SLOs. The component is executed each time an (i) interaction (i.e., read, write, or update) with a data object takes place, (ii) the set of available storages changes, and (iii) in predefined optimization intervals. During the optimization, the component finds the cheapest placement solution for a data object, respectively its data object chunks. Furthermore, the optimization guarantees that the SLOs availability, durability and vendor lock-in are not violated.

By applying the assumption that the usage pattern of a data object chunk stays the same in the nearer future [16], the optimization process applies historical usage information to predict the future usage of a data object chunk. By optimizing only the placement of the data object chunk of one data object, CORA is able to handle a huge amount of objects, while by applying a global optimization for all objects at the same time, the optimization duration increases non-linearly with the amount of objects [17]. To consider also not used objects, CORA regularly performs an optimization that includes not or rarely used data object chunks. By applying this regular optimization, such chunks are migrated to long-term storage systems. After the optimization is done, the transfer to and from the storages, and the erasure coding mechanism is handled in a separate thread so that a new optimization, from a new request, can be handled in parallel.

The historical access and storage information of each data object is stored in a local *Database*. Finally, the *API* component is responsible for the interaction between the user and the CORA middleware. The *Transfer Handler* is responsible for the interaction with the cloud providers.

Since CORA features a loosely coupled architecture, each component could be substituted. In future revisions of CORA, especially, arbitrary optimization approaches can be integrated and used as foundation for the Optimizer. In the next section, we will formulate the MILP-based optimization approach currently applied in CORA.

IV. Data Object Placement

In the following we discuss the applied system model and the formal specification of the optimization model.

A. System Model

As already defined, our optimization suggests the placement of a data object, or more precisely for data object chunks, on several cloud storage providers in the most cost-efficient way without violating the defined SLOs. For this optimization, we provide a MILP-based data placement approach. In the following, we will introduce the used variables, the cost model, and the used decision variables, before we will discuss the objective function in greater detail.

1) Variables: In our model, the set of available storage systems is labeled with S, where $s \in S = \{s_1, s_2, \ldots\}$ indicates a specific storage. Furthermore, the data object is labeled with F. $f \in F = \{f_1, f_2, \ldots\}$ indicates a specific data object chunk, where $|S| \geq |F|$. The amount of chunks depends on the erasure coding configuration, e.g., for an erasure coding of (3,4) there will be four data object chunks; each chunk belongs to one data object. For each data object chunk, information about past data access (i.e., size changes, used incoming and outgoing traffic and amount of operations) is stored in the Database. Under the assumption that the usage pattern of a data object stays the same over a period of time [16], this historical time steps are analyzed to predict the future usage of each data object chunk. The parameter τ defines the amount of historic information that is taken into account for the optimization, e.g., τ = 300 minutes means that the last 5 hours of access history is considered.

2) Cost Model: The cost that are billed to store a data object chunk, are composed of the used storage cost, data transfer cost and access operation cost. In addition, there may also be migration cost. Furthermore, if the chunk is stored on a long-term storage, the BTU and BSU have to be considered.

The overall cost calculation is shown in (1). This equation calculates the cost that occur if the data object chunk f is

stored at storage s by taking the usage history of the past τ minutes of f into account.

$$c_{(s,f,\tau)} = c^{S}_{(s,f,\tau)} + c^{R}_{(s,f,\tau)} + c^{W}_{(s,f,\tau)} + c^{T_{in}}_{(s,f,\tau)} + c^{T_{out}}_{(s,f,\tau)} \quad (1)$$

The equation to calculate the storage cost that occur to store a data object chunk f on storage s is shown in (2). The term $p^{S}_{(s,\gamma_{(s,f)})}$ calculates the storage price. Since many storage providers offer a price reduction that is correlated to the space consumption, this has to be taken into account in the calculations. To incorporate this price reduction, the cost model uses $\gamma_{(s,f)}$ to calculate the used space of the current billing period, e.g., a month. If a data object chunk f is currently not stored at storage s, f is added to the calculation of $\gamma_{(s,f)}$. This value is then multiplied with the data object chunk size, defined by $\sigma_{(f,\tau)}$. This term includes the last τ minutes of historic usage information of f. If the storage s has a BSU defined and the size of f is smaller than the BSU, the value of the BSU is used for $\sigma_{(f,\tau)}$.

If f is currently located on a long-term storage but the end of the BTU has not yet been reached, the remaining BTU cost need to be integrated into the calculations as well. This is done by $\hat{\sigma}_{(f,BTU)} \cdot v_{\hat{s}_f}$. The term $\hat{\sigma}_{(f,BTU)}$ returns the size of the data object chunk f that is charged for the remaining time until the end of the BTU. $v_{\hat{s}_f} \in \{0,1\}$ defines if a data object chunk is currently stored on a long-term storage ($v_{\hat{s}_f} = 1$) or not ($v_{\hat{s}_f} = 0$). It has to be noted that our approach only takes the storage that our system uses into account. If the customer uses the storage additionally to save other data our cost model does not include this used storage into the calculation.

$$c^{S}_{(s,f,\tau)} = p^{S}_{(s,\gamma_{(s,f)})} \cdot (\sigma_{(f,\tau)} + \hat{\sigma}_{(f,BTU)} \cdot v_{\hat{s}_f}) \quad (2)$$

The cost that occur due to the performed write operations is calculated by (3) and the read operation cost by (4). The values $r^{W}_{(f,\tau)}$, respectively $r^{R}_{(f,\tau)}$, consider the amount of performed write and read operations of data object chunk f in the last time period τ. The term p^{W}_{s} defines the price of n write operations; n is defined by $p^{W_{step}}_{s}$. The terms p^{R}_{s} and $p^{R_{step}}_{s}$ are the corresponding read values. Delete operations are handled analogously to (3) and (4). However, it should be noted that there are only few providers charging for deleting.

$$c^{W}_{(s,f,\tau)} = r^{W}_{(f,\tau)} \cdot \frac{p^{W}_{s}}{p^{W_{step}}_{s}} \quad (3)$$

$$c^{R}_{(s,f,\tau)} = r^{R}_{(f,\tau)} \cdot \frac{p^{R}_{s}}{p^{R_{step}}_{s}} \quad (4)$$

(5) and (6) provide the cost calculations for the performed incoming and outgoing transfer of data object chunk f. In the following, we will only discuss (5) in detail, since the description is analogously applicable for (6). In (5), $t^{out}_{(f,\tau)}$ defines the amount of bytes that were read from the provider during τ. Furthermore, $p^{T_{out}}_{(s,\beta_{(s,f)})}$ defines the outgoing transfer price for the storage. Same as for the storage cost, most providers offer price reductions for the transfer cost. Analogue to $\gamma_{(s,f)}$ in (2), $\beta_{(s,f)}$ calculates the amount of transferred bytes of the storage s in the current billing period, including the amount of transferred bytes of the current data object chunk f. If the storage is a long-term storage, additional data retrieval cost can be charged. The data retrieval price is defined by $p^{ret}_{(s,\beta_{(s,f)})}$ and $v_{\hat{s}_f}$ is the same as in (2).

$$c^{T_{out}}_{(s,f,\tau)} = t^{out}_{(f,\tau)} \cdot (p^{T_{out}}_{(s,\beta_{(s,f)})} + p^{ret}_{(s,\beta_{(s,f)})} \cdot v_{\hat{s}_f}) \quad (5)$$

$$c^{T_{in}}_{(s,f,\tau)} = t^{in}_{(f,\tau)} \cdot p^{T_{in}}_{(s,\beta_{(s,f)})} \quad (6)$$

If f has to be migrated from one storage to another, the occurring cost have to be included in the calculation as well. This calculation depends on the migration: If the providers of the source and destination storages are different, the migration cost is calculated by an addition of the outgoing cost of the source storage and the incoming cost of the destination storage. Those cost are calculated according to (7), where $\hat{\sigma}_f$ specifies the current size of the data object chunk and $p^{T_{out}}_{(s,\beta_{(s,f)})}$, respectively $p^{T_{in}}_{(s,\beta_{(s,f)})}$, are the same as in (5) and (6). Further, $r^{R}_{s_1}$ and $r^{W}_{s_2}$ specify the amount of required read and write requests. The terms p^{R}_{s}, $p^{R_{step}}_{s}$, p^{W}_{s} and $p^{W_{step}}_{s}$ are also the same as in (3) and (4). If f is stored on a long-term storage, the data retrieval cost has to be included. This is done by the term $p^{ret}_{(s,\beta_{(s,f)})} \cdot v_{\hat{s}_f}$ where $p^{ret}_{(s,\beta_{(s,f)})}$ and $v_{\hat{s}_f}$ are also the same as in (5). If the source and destination storages have the same provider, but the storages are in different regions and this provider offers a reduced migration price, (8) is applied. The first term, $p^{T_{(out,reg)}}_{s_1}$, defines the outgoing price from one region to another. The second term, $p^{T_{(in,reg)}}_{s_2}$, defines the incoming price. The remaining terms are the same as in (7).

$$\begin{aligned} c^{M}_{(s_1,s_2,f)} =& (p^{T_{out}}_{(s_1,\beta_{(s_1,f)})} + p^{T_{in}}_{(s_2,\beta_{(s_2,f)})} + p^{ret}_{(s,\beta_{(s,f)})} \cdot v_{\hat{s}_f}) \\ & \cdot \hat{\sigma}_f + r^{R}_{s_1} \cdot \frac{p^{R}_{s_1}}{p^{R_{step}}_{s_1}} + r^{W}_{s_2} \cdot \frac{p^{W}_{s_2}}{p^{W_{step}}_{s_2}} \end{aligned} \quad (7)$$

$$\begin{aligned} c^{M_{reg}}_{(s_1,s_2,f)} =& (p^{T_{(out,reg)}}_{(s_1,\beta_{(s_1,f)})} + p^{T_{(in,reg)}}_{(s_2,\beta_{(s_2,f)})} + p^{ret}_{(s,\beta_{(s,f)})} \cdot v_{\hat{s}_f}) \\ & \cdot \hat{\sigma}_f + r^{R}_{s_1} \cdot \frac{p^{R}_{s_2}}{p^{R_{step}}_{s_2}} + r^{W}_{s_2} \cdot \frac{p^{W}_{s_2}}{p^{W_{step}}_{s_2}} \end{aligned} \quad (8)$$

3) Decision Variables: To mark if a data object chunk f is stored on a particular storage s, we use the binary decision variable $x_{(s,f)} \in \{0,1\}$. $x_{(s,f)} = 1$ indicates that f is stored at s; $x_{(s,f)} = 0$ otherwise. Furthermore, the system model uses $g_{(\tilde{S},F)} \in \{0,1\}$ where $\tilde{S} = \{s_1, s_2, \ldots, s_n\}$ is the set of selected storages with $|\tilde{S}| = |F|$ and $\tilde{S} \subseteq S$. $g_{(\tilde{S},F)} = 1$ indicates that each storage in $\tilde{S}$ has one data object chunk of F stored. $g_{(\tilde{S},F)} = 0$ indicates that at least one of the storages in $\tilde{S}$ does not have a data object chunk of F stored.

The decision variables $z_{(s_1,s_2)}$ and $y_{(s_1,s_2)}$ specify if two storages are identical or do have the same storage provider but are different storage regions. $z_{(s_1,s_2)} \in \{0,1\}$ defines if two storages s_1 and s_2 are not identical and have different storage providers. $z_{(s_1,s_2)} = 1$ indicates that this is true, $z_{(s_1,s_2)} = 0$ otherwise. Analogously, $y_{(s_1,s_2)} \in \{0,1\}$ defines if two storages s_1 and s_2 are not identical, but have the same

storage provider. Finally, the decision variable $v_{\hat{s}_f} \in \{0,1\}$ defines if a data object chunk is currently stored on a long-term storage, indicated by $v_{\hat{s}_f} = 1$, or not, indicated by $v_{\hat{s}_f} = 0$.

B. Placement Problem

After defining the system model, we are now able to discuss the optimization problem. As already described, the optimization problem defines the problem of finding the most cost-efficient placement of each chunk of a data object F. Therefore, the optimization problem is provided with the data object F and the available storage providers S. Additionally, the problem takes the time period τ as input.

1) Objective Function: (9) shows the objective function, which is set to minimize the overall cost to store F:

$$\min \sum_{f \in F} \sum_{s \in S} \big(c_{(s,f,\tau)} \cdot w_{(s,f)} + c^{M}_{(\hat{s}_f,s,f)} \cdot z_{(\hat{s}_f,s)} + c^{M_{reg}}_{(\hat{s}_f,s,f)} \cdot y_{(\hat{s}_f,s)} \big) \cdot x_{(s,f)} \tag{9}$$

$c_{(s,f,\tau)} \cdot w_{(s,f)}$ calculates the overall cost to store the data object chunk f on the storage provider s by taking the last τ minutes of the data object chunk usage history information into account. The term $c_{(s,f,\tau)}$ was already discussed in Section IV-A2. The term $w_{(s,f)} \in [1, BTU]$ is a multiplier that helps to specify if the overall storage cost can be reduced by storing a data object chunk on a long-term storage or not. If s is a long-term storage, the calculation of the resulting value of the term is initialized with the value of the BTU, i.e., $w_{(s,f)} = BTU$. The algorithm decreases $w_{(s,f)}$ each time there was no or rare usage of the data object chunk according to the history information, where the amount of history information is defined by the BTU. This is done until all history information is checked or until $w_{(s,f)} = 1$. If s is a standard storage without BTU, the value is always 1.

$c^{M}_{(\hat{s}_f,s,f)} \cdot z_{(\hat{s}_f,s)}$ and $c^{M_{reg}}_{(\hat{s}_f,s,f)} \cdot y_{(\hat{s}_f,s)}$ calculate the migration cost from one storage to another, without and with special migration prices. Finally, the decision variable $x_{(s,f)}$ decides if the data object chunk f is stored on storage s or not.

2) Constraints: Constraint (10) ensures that the vendor lock-in factor l_F, which was defined by the owner of the data object as an SLO, is fulfilled.

$$\frac{1}{\sum_{f \in F} \sum_{s \in S} x_{(s,f)}} \leq l_F \tag{10}$$

Further constraints ensure that the required availability (11) and durability (12) are fulfilled. Since (11) and (12) are defined in a similar manner, we only discuss the former:

$\sum_{\tilde{S}' \in v_{(\tilde{S},k)}} \big[\prod_{s \in \tilde{S}'} \hat{a}_s \cdot \prod_{s \in \tilde{S} \setminus \tilde{S}'} (1 - \hat{a}_s) \big]$ calculates the availability of the storage set $\tilde{S}'$. $\tilde{S}'$ holds all possible combinations of size k of the storage set $\tilde{S}$. Those combinations are represented by $v_{(\tilde{S},k)}$. The term $\hat{a}_s$ defines the availability of s. Conclusively, this part of the equation calculates the probability that there are k simultaneously available storages. To complete (11), we have to include the functionality that a system which uses erasure coding with a (m,n) coding configuration, can withstand up to $n-m$ simultaneous storage failures. This is done by increasing k starting from m (i.e., the amount of minimum required data object chunks of F depicted by F^m) to $|\tilde{S}|$. In the equation this is achieved by $\sum_{k=|F^m|}^{|\tilde{S}|}$.

$$\sum_{k=|F^m|}^{|\tilde{S}|} \sum_{\tilde{S}' \in v_{(\tilde{S},k)}} \Big[\prod_{s \in \tilde{S}'} \hat{a}_s \cdot \prod_{s \in \tilde{S} \setminus \tilde{S}'} (1 - \hat{a}_s) \Big] \geq a_F \cdot g_{(\tilde{S},F)} \tag{11}$$

$$\sum_{k=|F^m|}^{|\tilde{S}|} \sum_{\tilde{S}' \in v_{(\tilde{S},k)}} \Big[\prod_{s \in \tilde{S}'} \hat{d}_s \cdot \prod_{s \in \tilde{S} \setminus \tilde{S}'} (1 - \hat{d}_s) \Big] \geq d_F \cdot g_{(\tilde{S},F)} \tag{12}$$

Finally, the calculated value is compared to $a_F \cdot g_{(\tilde{S},F)}$. a_F defines the owner-defined required data object availability of the data object. $g_{(\tilde{S},F)}$ defines if each storage in $\tilde{S}$ has one data object chunk stored or not. $g_{(\tilde{S},F)}$ is defined by constraints (13) and (14), with $i \in \{1, \ldots, |\tilde{S}|\}$.

$$g_{(\tilde{S},F)} \geq \sum_{s \in \tilde{S}} \sum_{f \in F} x_{(s_n,f)} - (|F| - 1) \tag{13}$$

$$g_{(\tilde{S},F)} \leq \sum_{f \in F} x_{(s_i,f)} \tag{14}$$

Furthermore, constraint (15) ensures that only $|F|$ assignments from data object chunks $f \in F = \{f_1, f_2, \ldots\}$ to the storages $s \in S = \{s_1, s_2, \ldots\}$ exist. (16) ensures that each data object chunk is stored at only one storage.

$$\sum_{f \in F} \sum_{s \in S} x_{(s,f)} = |F| \tag{15}$$

$$\sum_{s \in S} x_{(s,f)} \leq 1; \ \sum_{f \in F} x_{(s,f)} \leq 1 \tag{16}$$

V. Evaluation Setup

To evaluate our optimization approach and the proposed middleware CORA, both have been prototypically implemented. For the evaluation, we use a real world cloud storage access trace [18].

A. Prototype

The prototype is implemented in Java and uses CPLEX[5] to solve the optimization problem, Apache jclouds[6] to connect to different storages and an erasure coding library[7] that implements the Reed-Solomon codes [11].

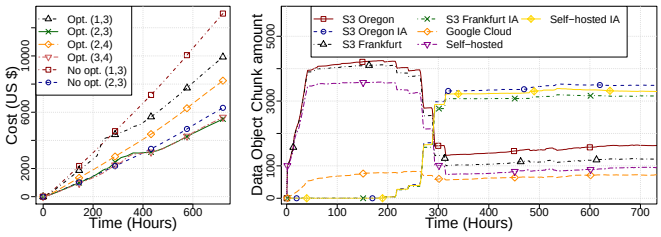
B. Storages

In the evaluation, we evaluate and analyze the behavior of our optimization with real world cloud storage systems. Beside the public cloud storage solutions from Amazon (AWS S3) and Google (Google Cloud Storage), we use a self-hosted Swift[8] storage system. Table II provides an overview of the cloud storages used for the evaluation.

[5] http://www.ibm.com/software/commerce/optimization/cplex-optimizer/
[6] https://jclouds.apache.org
[7] https://www.backblaze.com/blog/reed-solomon/
[8] https://wiki.openstack.org/wiki/Swift

TABLE II
EVALUATION STORAGES

Provider	Region	Storage Class
AWS S3	US Oregon	Standard Storage
AWS S3	US Oregon	Infrequent Access (IA)
AWS S3	US North California	Standard Storage
AWS S3	EU Frankfurt	Standard Storage
AWS S3	EU Frankfurt	Infrequent Access (IA)
AWS S3	Asia Pacific Tokyo	Standard Storage
AWS S3	Sao Paulo	Standard Storage
Google Cloud	-	Standard Storage
self-hosted	-	Standard Storage
self-hosted	-	Long-term Storage



(a) Cost (b) Data object chunk distribution

Fig. 2. With long-term storage evaluation results.

The evaluation uses the pricing models from AWS S3[9], respectively Google Cloud Storage[10]. For the self-hosted standard storage we apply the AWS S3 Frankfurt standard storage pricing model and for the self-hosted long-term storage the AWS S3 Frankfurt IA pricing model.

C. Evaluation Data

For the evaluation, we use an access trace from the public available dataset discussed in [18]. This dataset contains 30 days of anonymised data objects access information on cloud storages used by $>$ 1,000,000 users.

For our evaluation, we use the access logs of 5,021 data objects of this trace. To include heavily used data objects, as well as seldom used data objects, the data objects are selected equally among all data objects of the dataset. Besides the normal usage of the cloud storage by the user, the dataset also includes three DDOS attacks to the storage. During those attacks the usage of the storage increases drastically. In our evaluation we did not include those attacks, because we only consider the normal usage of a cloud storage at the moment.

The used dataset also includes data objects that were uploaded to the storage before the trace was created, but read during this time. For those data objects we upload the data objects at the beginning of the evaluation to make sure that the data objects are available at the time when they are used.

As SLOs, we define that each data object has to be stored with, at least, an availability of 99.99%, a durability of 99.9999999% and a vendor lock-in factor of 0.5.

D. Evaluation Process

During the evaluation, we iterate through the access traces of the 5,021 selected data objects and perform the logged operations (read, write, and delete).

All evaluations use the entire 30 days of the trace. To evaluate the complete behavior of our optimization we set the billing period and the BTU to one week. Further, the prototype generates a new history time step every 30 minutes and the optimization analyzes the last 5 history steps.

During the evaluation we also log the duration of the optimizations and the amount of used metadata, to be able evaluate the effort, in terms of time and memory, that our optimization approach requires.

[9]https://aws.amazon.com/s3/pricing/

[10]https://cloud.google.com/storage/pricing

E. Baseline

As a baseline for each scenario, the cost are calculated that would have been charged if the data objects would have been uploaded and accessed without our optimization. For this we disable the optimization and define a fixed set of storages that is then used equally by the middleware.

VI. EVALUATION SCENARIOS

A. With Long-Term Storages

Within this evaluation scenario, we evaluate how our optimization handles the placement of the data objects when long-term storages are available and different erasure coding configurations are used. This scenario includes the complete storage set from Table II. For the optimization evaluation, CORA is configured to perform an optimization of all not used data objects every 8 days.

Evaluation Hypotheses: At the beginning the optimization should select the cheapest standard storages. After each 8 days all not used data object chunks should be transferred to long-term storages. Ideally, the data object chunks that are transferred to long-term storages should be well placed and stay unused for a longer time.

Baseline: As baselines, we use two fixed storage provider subsets that include only standard storages. The first subset is: AWS S3 US Oregon standard storage, AWS S3 EU Frankfurt standard storage and the self-hosted standard storage. Further, this baseline uses the erasure coding configuration (2,3). The second subset contains the most expensive storages of our storage set. This storage subset contains: AWS S3 Tokyo standard storage, AWS S3 Sao Paulo standard storage and the AWS S3 North California standard storage. This baseline uses the erasure coding configuration (1,3).

Evaluation Execution: Figure 2 shows the results of this evaluation. Figure 2a shows the cumulative cost that would have been charged with and without our optimization and with different erasure coding configurations. Figure 2b shows the distribution of the data object chunks on the storages during the evaluation with an active optimization and the erasure coding configuration (2,3). For a clearer graph the storages that are not selected by the optimization are omitted.

As can be seen, at the beginning, the cost for all approaches increases steadily. The cost of the second baseline, i.e., the expensive storages, increases most rapidly. The first baseline

and the runs using active optimization and an (2,3) erasure coding configuration, respectively (3,4), are the ones with the slowest cost increase.

This cost difference is, besides the usage of different storages, due to the different m values of the (m,n) erasure coding configurations. By storing a data object with a (1,3) configuration the data object is stored by replication of the data object chunk, as described in Section II-B. A (2,3) configuration stores the data in a way that two of three data object chunks are needed to restore the data object and, thus, the data object chunks are smaller than for a (1,3) configuration [14].

After 192 hours, the optimization of the not used data objects took place. This optimization step transferred several not used data object chunks to long-term storages. This can be observed in Figure 2b, where several data object chunks are copied from AWS S3 US Oregon standard storage, AWS S3 EU Frankfurt standard storage and the self-hosted standard storage to AWS S3 US Oregon IA storage, AWS S3 US Frankfurt IA storage and the self-hosted long-term storage. Figure 2a shows that this optimization of the not used data object chunks increases the overall cost for the executions with active optimization. This is due to the fact that the long-term storages charge for the whole BTU as soon as a data object chunk is stored on it. Nevertheless, this also means that as long as the BTU is not over, no additional storing cost for the long-term storages are added. Due to the huge amount of data object chunks that has to be optimized and migrated, including the need to process customer issued read and write requests in between, this process takes time and is done after 312 hours.

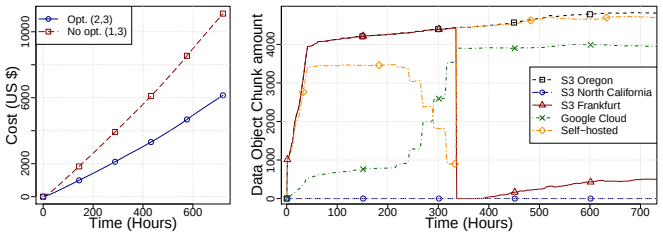
After 480 h, all additional cost due to the BTU are charged and the storage cost of the long-term storages are added and, thus, the graph rises faster again. However, the executions with active optimization and the erasure coding configuration (2,3) and (3,4) are already cheaper than the baseline.

After the BTU is over, it can be seen that all cost rises steadily without bigger changes. Also in Figure 2b no big changes happen. This is due to the fact that the data object chunks, which were selected by our optimization for the long-term storage, are well placed and are still not used.

Results: Altogether, with the help of our optimization we were able to save 12.61% of the cost in comparison to the first baseline and 57.59% of the cost in comparison to the second baseline. Furthermore, we showed that the optimization selects the data object chunk placement, especially the not or rarely used chunks, in a well placed manner. Consequently, for longer evaluation runs, the cost savings would be even better in comparison to the baseline, due to the linear cost increase after the optimization of the not used files.

B. Temporary Unavailable Storage

The aim of this scenario is to evaluate the behavior of the optimization if a storage is temporarily not available. For this, we do not include the long-term storages. At the beginning of this evaluation, the storage set includes all standard storages. After 14 days, the AWS S3 Frankfurt storage is taken from the storage set, simulating the unavailability of it.



(a) Cost (b) Data object chunk distribution

Fig. 3. Storage unavailability and repair functionality.

Evaluation Hypotheses: The unavailability of the storage should lead to the repair process that recreates the missing data object chunks, the chunks that were stored on AWS S3 Frankfurt, by downloading the remaining available chunks and use them to recreate the missing ones. The optimization should then select a new storage solution for the data object chunks. After further 48 hours, the AWS S3 Frankfurt storage is added to the storage set again, which simulates the situation where the storage is available again. This should lead to a re-optimization process with all available storages.

Baseline: As a baseline, we use the following fixed storage subset: AWS S3 US Oregon standard storage, AWS S3 US Frankfurt standard storage and the self-hosted standard storage. The baseline uses a (1,3) erasure coding configuration.

Evaluation Execution: Figure 3 shows the results of this evaluation. Figure 3a shows the cumulative cost of the baseline and the evaluation with the active optimization. Figure 3b shows the distribution of the data object chunks on the storages during the evaluation with the active optimization.

As can be seen in Figure 3b, after 336 hours the AWS S3 Frankfurt storage is removed from the storage set, which leads to a loss of all data object chunks that are stored on this storage. Furthermore, it can be observed that simultaneously several data object chunks are stored at the self-hosted storage and the Google Cloud storage. Those two storages were selected by the optimization as a replacement of the unavailable AWS S3 Frankfurt storage, so that the required redundancy and availability is restored again.

After another 48 hours the AWS S3 Frankfurt storage is available again. The first step after a storage is available again, is to delete all data object chunks on that storage which are now inconsistent. Following to this step the optimization starts to use the storage again, which can be observed in Figure 3b.

In the case of the baseline, it can be seen that the cumulative cost increases faster than for the evaluation run with active optimization. This is due to the bigger data object chunks of the (1,3) erasure coding configuration, which are three identical replications of the original data object. Moreover, due to the missing optimization which is needed by the repair functionality, the system has only two replications left, instead of three, after the storage is unavailable.

Results: For this evaluation scenario we were able to show that our approach guarantees the redundancy and availability of the data, even if a storage is temporarily unavailable.

TABLE III
AVERAGE OPTIMIZATION DURATIONS IN MILLISECONDS (STANDARD DEVIATION)

Period of time	Erasure Coding configurations			
	(1,3)	(2,3)	(2,4)	(3,4)
2014-01-11 - 2014-01-16	65.73 (σ = 19.85)	70.44 (σ = 21.04)	145.12 (σ = 35.00)	147.40 (σ = 35.41)
2014-01-16 - 2014-01-21	89.00 (σ = 18.05)	96.68 (σ = 20.36)	187.18 (σ = 33.49)	180.20 (σ = 28.87)
2014-01-21 - 2014-01-26	226.03 (σ = 74.23)	220.29 (σ = 67.11)	394.10 (σ = 108.53)	347.91 (σ = 107.42)
2014-01-26 - 2014-01-31	214.81 (σ = 27.03)	227.88 (σ = 29.14)	296.43 (σ = 40.93)	287.29 (σ = 40.26)
2014-01-31 - 2014-02-05	209.32 (σ = 52.69)	195.79 (σ = 44.52)	332.59 (σ = 87.71)	324.72 (σ = 64.01)
2014-02-05 - 2014-02-10	244.48 (σ = 37.03)	320.39 (σ = 57.32)	371.93 (σ = 62.35)	373.69 (σ = 51.28)
2014-02-10 - 2014-02-11	295.73 (σ = 27.44)	255.57 (σ = 22.65)	300.89 (σ = 33.31)	383.67 (σ = 36.13)

Furthermore, it can be seen that, even without the usage of long-term storages, our approach saved 44.64% of the cost compared to the baseline.

C. CORA Performance Assessment

In the following we analyze the performance of CORA, by analyzing the optimization duration and amount of metadata that is used during the evaluation.

Duration: During the execution of the first evaluation scenario (see Section VI-A) we logged the duration of each execution of the optimization. Table III shows the average duration of the optimizations, including the standard deviation, in milliseconds. As can be observed, for all erasure coding configurations, the execution durations start with lower durations, e.g., 65.73 ms for a (1,3) configuration, and then increase. In the beginning the durations increase faster, but after one week of evaluation the durations are getting constant.

This is due to the fact that the complexity of the optimization problem depends on the amount of data object chunks and the amount of history information. At the beginning there are less data object chunks in the database and the data object chunks only have a small amount of history information. Therefore, the optimization can quickly process the data. During the runtime, the amount of data object chunks and the history information of them increase and, thus, the optimization duration increases as well. However, for the evaluation we set the billing period and the BTU to one week. Therefore, the optimization only processes the history information of one week and so the average duration, gets constant after some time.

Metadata: Our optimization and the proposed middleware produces and uses metadata, like the history information of the data object chunks, which are stored in the database. However, the optimization only needs the metadata of the last billing period and BTU time for the optimization, therefore CORA prunes the metadata. For example, for the first evaluation scenario (see Section VI-A) the size of the metadata after the execution with active optimization and an erasure coding configuration of (2,3) was 53MB and after the execution with an (3,4) configuration the size was 71MB.

VII. RELATED WORK

In recent years, the redundant storage of data in the cloud has been a vivid field of research. Substantial efforts have been undertaken, however, with some important limitations.

Similar to our own work, *Scalia* aims at minimizing the cost for redundant data storage in the cloud [17]. To achieve this, the system focuses on performing a runtime analysis of the access patterns of the data objects and uses this information to adapt data placement. For that, the system holds historical access information, e.g., the size of a data object chunk or input and output traffic for each data object, which are then used in a placement algorithm. Similar to CORA, Scalia applies erasure coding. In contrast to our work, where an optimal solution to the data placement problem is computed using MILP, Scalia relies on a heuristic to find a cost-efficient data placement solution. The heuristic resembles the well-known multi-dimensional knapsack problem. In addition, Scalia does not include long-term storage solutions and therefore does not recognize BTUs, i.e., the minimum storage duration for cloud storages is not regarded. Further, Scalia also does not include the Block Rate Pricing models of some providers. Instead, it uses a simplified pricing model. As a result, the pricing model applied by Scalia is not completely realistic. While Scalia applies heuristics and does not take into account the BTU, it nevertheless comes closest to our own work.

Similar to Scalia, *RACS* uses erasure coding to split data objects into several data object chunks and to store them on several cloud providers [6]. In contrast to Scalia and CORA, RACS does not monitor the usage of the data objects. Hence, RACS is not able to take this information into account for finding a cost-efficient data placement.

Another cost-efficient multi-cloud storage system is *CHARM* [9]. Similar to CORA, CHARM offers the functionality to find the cheapest storage solution from a set of available cloud storage providers to offer a high availability and to avoid vendor lock-in. However, in comparison to CORA the system uses two separate redundancy mechanisms, *replication* and *erasure coding*. The system uses the access history of a data object to determine if the storage cost are lower for one of these two mechanisms. CHARM uses a similar pricing model as Scalia, leading to the same limitations.

MetaStorage uses full replication to store data objects on several cloud storage providers aiming at a high data availability [19]. To distribute data objects among the available providers, the system uses a distributed hashtable, which makes the MetaStorage highly scalable. In contrast to our work, MetaStorage does not include any optimization of the placement to find the cheapest provider set. Furthermore, all data objects are fully replicated among the different storage

providers. This redundancy mechanism raises the amount of needed traffic and storage and therefore increases the cost.

While storage cost are regarded in further approaches [20], [21], data transfer cost are not regarded. *HAIL* [22] does not take into account any cost for storing and transferring data objects. Beside the usage of a redundant storage functionality in the cloud, there are also several Peer-to-Peer (P2P) systems that are offering similar functionalities, like [23], [24]. The usage of different redundancy mechanisms, i.e., replication and erasure coding, were also analyzed in P2P [13], [14].

Apart from Scalia and CHARM, none of the abovementioned approaches provides a cloud-based redundant storage system that monitors the usage of the data objects and dynamically optimizes the placement of the data objects in a cost-efficient way while taking into account SLOs. To the best of our knowledge, none of the discussed approaches includes the usage of long-term storages to store not or rarely accessed data objects. Therefore, state-of-the-art solutions, as discussed above, do not recognize BTUs and BSUs. Last but not least, none of the discussed works models the problem using MILP.

VIII. Conclusions

Using several cloud-based storage systems to store data in a reliable, redundant and cost-efficient way seems to be an obvious choice in order to avoid vendor lock-in. Within this work, we formulate a data object placement model that is able to optimize the storage of data objects on several cloud storage providers in a cost-efficient and redundant way. The optimization, further, ensures that predefined storage requirements of the customer are fulfilled at any time. Furthermore, we presented *CORA*, a framework that uses MILP to solve the optimization problem. In the end, we thoroughly evaluated the approach with a real world access trace and compared the results with placement solutions without an optimization but with a fixed provider set.

In our evaluation we were able to show that our optimization approach is able to provide a much cheaper solution, compared to the baselines. Nevertheless, due to the high complexity of our optimization we are planning to analyze different heuristic approaches to solve the optimization problem. Further, we will extend the architecture of CORA to a distributed architecture, to be scalable in terms of throughput. Additionally, we want to evaluate different approaches to predict the future data object access more precisely. Besides the analysis of historic usage information to optimize the placement (as applied in this paper), this extension will also include approaches for the prediction of future data access.

Acknowledgment

This work is partially supported by the Commission of the European Union within the CREMA H2020-RIA project (Grant agreement no. 637066) and by TU Wien research funds.

References

[1] E. Allen and C. M. Morris, "Library of congress and duracloud launch pilot program using cloud technologies to test perpetual access to digital content," in *Library of Congress, News Release*, July 14 2009, Accessed: October 2015.

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

[3] B. Butler, "Gartner: Top 10 cloud storage providers," http://www.networkworld.com/article/2162466/cloud-computing/cloud-computing-gartner-top-10-cloud-storage-providers.html, Accessed: Oct. 2015.

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

[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 *1st ACM Symp. on Cloud Computing*, 2010, pp. 229–240.

[7] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," *Comm. of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[8] G. Vernik, A. Shulman-Peleg, S. Dippl, C. Formisano, M. C. Jaeger, E. K. Kolodner, and M. Villari, "Data on-boarding in federated storage clouds," in *6th Int. Conf. on Cloud Comp.*, 2013, pp. 244–251.

[9] Q. Zhang, S. Li, Z. Li, Y. Xing, Z. Yang, and Y. Dai, "CHARM: A Cost-Efficient Multi-Cloud Data Hosting Scheme with High Availability," *IEEE Trans. on Cloud Computing*, vol. 3, no. 3, pp. 372–386, 2015.

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

[11] J. S. Plank, "Erasure codes for storage systems: A brief primer," *Login: The USENIX Magzine*, pp. 44–50, 2013.

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

[13] R. Rodrigues and B. Liskov, "High availability in DHTs: Erasure coding vs. replication," in *4th Int. Conf. on P2P Systems*, 2005, pp. 226–239.

[14] H. Weatherspoon and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Revised Papers from the First Int. Workshop on P2P Systems*, 2002, pp. 328–338.

[15] M. Naldi and L. Mastroeni, "Cloud storage pricing: a comparison of current practices," in *2013 Int. Workshop on Hot Topics in Cloud Services*, 2013, pp. 27–34.

[16] A. Mahanti, D. Eager, and C. Williamson, "Temporal locality and its impact on web proxy cache performance," *Performance Evaluation*, vol. 42, no. 2, pp. 187–203, 2000.

[17] T. G. Papaioannou, N. Bonvin, and K. Aberer, "Scalia: An adaptive scheme for efficient multi-cloud storage," in *Int. Conf. on High Performance Comp., Networking, Storage and Analysis*, 2012, pp. 20:1–20:10.

[18] R. Gracia-Tinedo, Y. Tian, J. Sampé, H. Harkous, J. Lenton, P. García-López, M. Sánchez-Artigas, and M. Vukolic, "Dissecting ubuntuone: Autopsy of a global-scale personal cloud back-end," in *Proc. of the 2015 ACM Conf. on Intern. Measurement Conf.* ACM, 2015, pp. 155–168.

[19] D. Bermbach, M. Klems, S. Tai, and M. Menzel, "MetaStorage: A Federated Cloud Storage System to Manage Consistency-Latency Tradeoffs," in *IEEE Int. Conf. on Cloud Computing*, 2011, pp. 452–459.

[20] C.-W. Chang, P. Liu, and J.-J. Wu, "Probability-based cloud storage providers selection algorithms with maximum availability," in *2012 41st Int. Conf. on Parallel Processing*, 2012, pp. 199–208.

[21] Y. Mansouri, A. N. Toosi, and R. Buyya, "Brokering algorithms for optimizing the availability and cost of cloud storage services," in *5th Int. Conf. on Cloud Comp. Tech. and Science*, 2013, pp. 581–589.

[22] K. D. Bowers, A. Juels, and A. Oprea, "HAIL: A high-availability and integrity layer for cloud storage," in *16th ACM Conf. on Computer and Communications Security*, 2009, pp. 187–198.

[23] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris, "Efficient replica maintenance for distributed storage systems," in *3rd Conf. on Networked Systems Design & Implementation*, 2006, pp. 45–58.

[24] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with cfs," in *18th ACM Symp. on Operating Systems Principles*, 2001, pp. 202–215.