Smart-Transfer: A Cost-Minimized Inter-Service Data Storage and Transfer Scheme

Galen Deal

A Thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2017

Thesis Committee:

Yang Peng, Committee Chair

Erika Parsons, Committee Member

Dong Si, Committee Member

Program Authorized to Offer Degree:
Computer Science and Software Engineering

©Copyright 2017
Galen Deal

University of Washington

Abstract

Smart-Transfer: A Cost-Minimized Inter-Service Data Storage and Transfer Scheme

Galen Deal

Chair of the Supervisory Committee:
Assistant Professor Yang Peng
Computing & Software Systems

Cloud storage services are a widely used tool in both industry and research. However, the wide variety of services offered by cloud providers raises the question of which storage services can most cheaply meet the needs of an application, particularly in cases where the performance required by that application varies over time.

To address this problem, we propose Smart-Transfer, a unique scheme to reduce the long-term cost of storing large data sets using cloud storage services. In contrast to most existing works, we study the storage cost-minimization problem by leveraging various available storage services that can provide different levels of performance at different pricing cost, under the constraint of data-access performance requirement. The key idea of Smart-Transfer is to continually transfer large data sets between different cloud storage services so as to always make the data ready within a specified period of time in at least one service that can satisfy the performance requirement while avoiding overpaying for unnecessarily high performance guarantees. While the selected storage service to satisfy a particular data-access request may not be the cheapest, the accumulative data-transfer and data-storage cost over a long period of time to satisfy a sequence of data-access requests can be minimized for the system. Simulation results show that Smart-Transfer performs well under various system parameters and request patterns, and can significantly reduce the cost compared to other schemes.

TABLE OF CONTENTS

Page

List of Figures iii

List of Tables v

Chapter 1: Introduction 1
- 1.1 Cloud Storage Services 1
- 1.2 Project Overview 3
- 1.3 Overview of Thesis 5

Chapter 2: Related Work 6
- 2.1 Cloud Service Aggregration 6
- 2.2 Cost Minimization 8

Chapter 3: Problem Description 10
- 3.1 System Model 10
- 3.2 Preprocessing 11
- 3.3 Problem Statement 17

Chapter 4: Design of Smart-Transfer Scheme 22
- 4.1 General Dynamic Programming Solution 22
- 4.2 Variations 24

Chapter 5: Performance Evaluation 31
- 5.1 Simulation Setup 31
- 5.2 Simulation Results 32

Chapter 6: Future Work 39

Chapter 7: Conclusion 42

Bibliography 44

LIST OF FIGURES

Figure Number Page

3.1 Result of preprocessing to fill in request gaps. For any time window for which no request originally exists, a new request is created for that time window with a performance requirement of 0 (i.e., it can be fulfilled by any storage service). 13

3.2 Result of preprocessing to remove redundant requests. In this case, request j is fully covered by request i (which has a higher performance requirement), and so can be removed. 14

3.3 Result of preprocessing to split up a request which is partially covered by a smaller request with a higher performance requirement. 15

3.4 Result of preprocessing to shorten a request which is partially overlapped by a request with a higher performance requirement. 16

3.5 Result of preprocessing to remove a request which is shorter than the data transfer delay and which is both preceded and followed by requests with higher performance requirements. In this case, the work of that request is taken over by the request which immediately precedes the short request. 17

3.6 Result of preprocessing to extend a request which is shorter than the data transfer delay and which is both preceded and followed by requests with lower performance requirements. In this case, $\tau = 2$. 18

3.7 Result of preprocessing to extend a request which is shorter than the data transfer delay and which is preceded by a request with a lower performance requirement and followed by a request with a higher performance requirement. In this case, $\tau = 2$. 19

4.1 Flowchart depicting Algorithm 1. 24

4.2 Flowchart depicting Algorithm 2. 25

5.1 Total cost on data storage and transfer under different numbers of total available storage services: the average request duration is 15 time units, the idle percentage is 0, the ratio of data-transfer cost to data-storage cost is 15, and the total simulation time is $43,200$ time units. 33

5.2 Total cost on data storage and transfer under different ratios of data-transfer to data-storage cost: 5 storage services are available, the average request duration is 15, and the total simulation time is 43,200 time units. The cost for the All-Transfer solution is only displayed when it is lower than that of the No-Transfers solution. 34
5.3 Improvement ratio of Smart-Transfer to the All-Transfer or No-Transfer scheme that yields a lower cost: 5 storage services are available, the average request duration is 15 time unites, and the total simulation time is 43,200 time units. 35
5.4 Total cost on data storage and transfer under different average request durations: 5 storage services are available, the ratio of data-transfer cost to data-storage cost is 12 for 5.4(a) and 28 for 5.4(b), and the total simulation time is 43,200 time units. 36
5.5 Improvement ratio of Smart-Transfer to the All-Transfer or No-Transfer scheme that yields a lower cost: 5 storage services are available, the ratio of data-transfer cost to data-storage cost is 12 for 5.5(a) and 28 for 5.5(b), and the total simulation time is 43,200 time units. 37
5.6 Performance under a 24-hour cycle of high and low loads: 5 storage services are available, the ratio of data-transfer cost to data-storage cost is 28, and the total simulation time is 43,200 time units. 38
5.7 Increased cost percentage resulting from splitting a month of request prediction window into short prediction windows: 5 storage services are available, the ratio of data-transfer cost to data-storage cost is 28, and the average request duration is 15 minutes. 38

LIST OF TABLES

Table Number Page

1.1 Examples of Cloud Storage Services 1
1.2 Performance Comparison of Cloud Storage Services 2
1.3 Price Factors of Cloud Storage Services 3
3.1 Summary of Notations 21

ACKNOWLEDGMENTS

I would like to thank the chair of my thesis committee, Doctor Yang Peng, for helping me throughout the capstone process. This thesis would not have been possible without his guidance, assistance, and hard work.

I would also like to thank the other members of my thesis committee, Doctors Erika Parsons and Dong Si, for their work in reviewing my thesis.

Finally – and most importantly – I would like to thank my parents, without whom nothing I do would be possible.

Chapter 1
INTRODUCTION

Cloud storage services are a valuable and widely used resource used in high volume by many different applications. In addition to its use in many areas of academic research [22–26, 44], cloud storage is being adopted at high rates by industry enterprises [34,37,39,40,43,66]. There are currently numerous cloud service providers which offer various types of unique cloud storage services with a range of competitive prices and performance guarantees (i.e., Service Level Agreements) to attract application developers. To demonstrate the variety of these offerings, a brief overview of some of the available cloud storage services follows.

1.1 Cloud Storage Services

Three of the most prominent cloud storage providers on the market today are Microsoft [50], Amazon [20], and Google [28], and each has its own suite of available cloud storage services.

Table 1.1 shows examples of available data storage services offered by major cloud service providers.

Table 1.1: Examples of Cloud Storage Services

provider	storage services
Microsoft	Blob, Queue, Table, File, DocumentDB, Redis Cache
Amazon	S3, Glacier, EBS, EFS, DynamoDB, ElastiCache
Google	Storage, Bigtable, Datastore

In contrast with the relatively stable pricing and performance structures of cloud storage services, data that makes regular use of these services may have time-varying levels of per-

formance requirements. For example, in the morning, the data about yesterday's news may be highly demanded by users in different regions and therefore the corresponding news data must be made ready in different storage services running in national or global data centers to guarantee the data availability and access latency. At midnight, however, a lower level of demand may be present and as such the performance requirements that must be met may be significantly lower. Over time, old news data may be only infrequently queried by a very small number of people.

Given the availability of compelling cloud storage services, it is wise for application developers or data service providers (e.g., media data companies and IoT data companies) to make efficient and economical usage of these services based on the time-varying performance requirement of the data saved in the cloud. Table 1.2 compares the performance levels of different cloud storage services, while Table 1.3 compares the different performance factors which affect the pricing of various cloud storage services.

Table 1.2: Performance Comparison of Cloud Storage Services

service		availability	latency	max request rate	max throughput
Azure	Blob [51, 57, 62]	$\geqslant$ 99.9%	unavailable	500/s	60 MB/s
	Table [52, 57, 62]	$\geqslant$ 99.9%	unavailable	20,000/s	2 MB/s
	Queue [53, 57, 62]	$\geqslant$ 99.9%	unavailable	2,000/s	2 MB/s
	File [54, 57, 62]	$\geqslant$ 99.9%	unavailable	unavailable	60 MB/s
	DocumentDB [56, 58, 60, 61]	$\geqslant$ 99.99%	unavailable	unlimited	ingress: 10 Gb/s egress: 20 Gb/s
	RedisCache [49, 59]	$\geqslant$ 99.9%	unavailable	$600 \sim 250{,}000$/s	$5 \sim 4{,}000$ Mb/s
AWS	S3 [16–18]	standard: 99.99% infrequent: 99.9%	milliseconds	unavailable	unavailable
	Glacier [15–18]	unavailable	minutes to hours	unavailable	unavailable
	EBS [6, 7]	99.999%	SSD tier: lowest HDD tier: higher	$250 \sim 20{,}000$ IOPS/volume	$250 \sim 500$ MB/s
	EFS [8–10]	high	low	unavailable	$>$ 10 GB/s
	DynamoDB [3–5]	high	$<$ 10 milliseconds	unlimited	unlimited
	ElastiCache [11–14]	high	low	extremely high	$<$ 10 GB/s
Google	Storage [30, 33]	$99\% \sim 99.95\%$	milliseconds	unavailable	unavailable
	BigTable [29, 31]	$\geqslant$ 99.5%	$<$ 10 milliseconds	10,000/s	10 MB/s/node
	Datastore [27, 32]	high	unavailable	unavailable	unavailable

Table 1.3: Price Factors of Cloud Storage Services

service		capacity	redundancy	requests	frequency	latency	capacity	throughput
Azure	Blob [51]	X	X	X	X			
	Table [52]	X	X	X				
	Queue [53]	X	X	X				
	File [54]	X	X	X				
	DocumentDB [56]	X		X				
	RedisCache [59]					X	X	X
AWS	S3 [16]	X		X	X			
	Glacier [15, 16]	X			X			
	EBS [6]	X		X		X		X
	EFS [8]	X			X			
	DynamoDB [5]	X		X				X
	ElastiCache [13]					X	X	X
Google	Storage [30]	X	X	X	X			
	BigTable [31]	X		X				X
	Datastore [32]	X		X				

1.2 Project Overview

This research project began as an investigation of minimal-cost strategies for using Microsoft's Azure Storage services to store IoT data in the cloud. One of the problems we encountered in this research was the question of which services should be used to store data, and when should they be used. Even when only considering Microsoft's storage services, there were multiple options with different levels of cost and performance. It was not at all obvious what the optimal strategy was to store the data for minimal cost while still meeting performance requirements. A search of the literature did not reveal any research which addressed this specific problem, indicating that this is an area of novel research. This research, then, is motivated by a desire to expand the current state of research in such a way as to address this problem, as well as to aid businesses who use existing cloud services to serve data to their users by helping them save money.

A simple solution would be to choose the storage service which is guaranteed to meet the highest level of performance requirement at all times. However, a potential problem arises if the requested data-access performance levels during the low-demand periods are significantly lower than those during the high-demand periods. In such cases, the business financing the application may be overpaying during the low-demand periods as they would still be paying

for the high performance guarantee needed to satisfy the high-demand periods.

The problem then is whether or not it is possible to make use of multiple storage services (of the same or different providers) so as to only utilize the higher-performance, more-expensive storage services when they are actually necessary and to avoid paying for them when they are not; and, if so, in what way should multiple storage services be used optimally in order to minimize costs while still meeting data-access performance requirements. As the movement of data between different storage services may incur high data-transfer costs, it is non-trivial to determine the cost-optimized way of moving and storing the data. When data needs to be transferred between regions or across providers, the cost may be even higher, which may limit the options given a certain budget.

To solve this problem, this thesis presents Smart-Transfer, a novel scheme which minimizes the cost of data transfer and storage in the cloud while still satisfying expected performance requirements of data accessibility. In this scheme, large data sets can be continually transferred between different storage services in the cloud so as to always make the data ready in at least one service that can satisfy the performance requirement of a data-access request. While the selected storage service used to satisfy a particular data-access request may not be the cheapest, the accumulative data-transfer and data-storage cost over a long period of time to satisfy a sequence of data-access requests can be minimized by Smart-Transfer.

In order to implement Smart-Transfer on top of existing cloud storage services, methods for transferring data between various services would be required. For data transfer between services from a single cloud storage provider, such providers often provide their own data movement and management services which could be used to accomplish this task. For example, Microsoft provides the Azure Data Factory service as part of its suite of Azure cloud services. Data Factory is, according to its product description, a "cloud-based data integration service that allows you to create data-driven workflows in the cloud for orchestrating and automating data movement" [55]. This service allows the user to schedule transfers of data from specified sources to specified destinations, and would be useful for executing the transfers suggested by Smart-Transfer provided these transfers are to and from storage des-

tinations which are supported by Data Factory. That is, once Smart-Transfer has generated its cost-minimized sequence of transfers, these transfers could be used as input for Data Factory, allowing Data Factory to actually execute the transfers. Similarly, Amazon provides the AWS Data Pipeline service which, according to their product description, allows users to "reliably process and move data between different AWS compute and storage services, as well as on-premise data sources, at specified intervals" [19]. These kinds of services which allow for automated, time-specified data transfers should be helpful in the implementation of Smart-Transfer using real-world storage services.

For data transfer between storage services from different cloud providers, some utility which is external to the providers themselves would likely be necessary. While such a system is outside the scope of this thesis, research has been done on the aggregation of disparate cloud services [2, 35, 36, 38, 45, 46, 65].

Extensive simulations have been conducted to evaluate the performance of the Smart-Transfer scheme. Results show that Smart-Transfer outperforms the compared schemes such as (i) maintaining data persistently in high-performance, high-cost storage services and (ii) continually moving data to storage services that can exactly satisfy each data-access request, under various system parameters.

1.3 Overview of Thesis

The rest of this thesis is organized as follows. Chapter 2 reviews related works. Chapter 3 formally describes the problem investigated in this thesis, presenting the system model and analytical study. Details of the Smart-Transfer scheme design are provided in Chapter 4. Chapter 5 shows the performance evaluation results obtained from simulations. Chapter 6 discusses potential avenues by which to extend this research. Finally, Chapter 7 concludes the thesis.

Chapter 2
RELATED WORK

Smart-Transfer is a novel scheme for data storage management, and as such there appears to be no existing research which fills the same role as this thesis. However, there does exist a large amount of research into various aspects of cloud storage, and much of it is related in some manner to the work presented in this thesis. The research which does relate to the work presented in this thesis generally falls into two different categories: research related to cloud service aggregation, and research related to cost minimization with respect to cloud resources.

2.1 Cloud Service Aggregration

Regarding aggregation of cloud services, there are many authors who have investigated this area of study with different focuses and goals. One common focus of this research is the issue of data availability. Mansouri, et al. [47], Abu-Libdeh, et al. [1], Wu et al. [67], and Liu, et al. [42] all investigate using multiple cloud providers to meet data availability requirements while minimizing costs. Abu-Libdeh, et al argue that striping data across multiple cloud storage providers can increase data availability and reliability and avoid vendor lock-in by reducing the cost of switching providers. In addition to advocating for such practices, the authors present RACS (Redundant Array of Cloud Storage), a prototype for a system which uses RAID-like techniques to replicate data across multiple cloud storage providers to achieve their goals of increasing availability and relieability and avoiding vendor lock-in.

Mansouri, et al. and Liu, et al. also investigate increasing availability by using multiple cloud storage providers. Perhaps more similarly to our research, both research groups present schemes for determining data replication strategies which minimize storage costs while still

meeting required levels of data availability. These schemes are then tested experimentally to show their effectiveness. While this research differs from our research in that it focuses on data availability as the primary requirement, it could potentially be of significant use should the work presented in this thesis be extended. Currently, the focus of Smart-Transfer is the minimization of costs while meeting general performance requirements, and as such does not take into account issues regarding data availability. However, availability is an important concern when implementing real-world applications. Should Smart-Transfer be incorporated into a real-world application, then making use of a scheme such as one of the ones proposed by Mansouri, et al. or Liu, et al. alongside Smart-Transfer could be helpful for addressing this concern.

Wu, et al. present a preliminary design for CSPAN, a system which synthesizes geographically distributed cloud storage services with the goals of minimizing costs, meeting latency requirements, and leveraging flexible consistency. This system could potentially address similar issues to the ones addressed by this thesis. However, the focus of CSPAN seems to primarily be data replication and the effective use of data centers distributed around the world.

The aggregation of cloud storage services is also explored by Machado, et al. [45,46], who present PiCsMu (Platform-independent Cloud Storage System for Multiple Usage), a system which aggregates heterogeneous cloud storage services in order to allow for the storage of arbitrary data in these potentially disparate storage systems. This work differs from ours in that it does not focus on cost minimization, but it is highly relevant to our work in that it demonstrates the feasibility of storing the same type of data in many different cloud storage services which were designed to accept specific types of data.

Extending the concept of cloud service aggregration to a more general level, Kash, et al. [38] and Jia, et al. [35,36] discuss the concept of a "supercloud," a cloud service collective made up of resources from heterogeneous cloud service providers. While the focus of this research is on virtual machines (VMs), it demonstrates the feasibility of building an overlay which can support user-level migration of VMs even between cloud services which do not

natively support user-level migration. Furthermore, it shows that this aggregation of services can be done in a cost-efficient manner, which is acutely relevant for the issues addressed in this thesis.

Finally, while it does not necessarily deal directly with cloud service aggregation, Globus Online [2] can potentially support such aggregation attempts by providing a method by which data can be migrated between cloud services. One of the fundamental aspects of the Smart-Transfer scheme presented in this thesis is the ability to move data from one cloud storage service to another. Globus Online provides a secure, simple, fault-tolerant method of file transfer between network-connected endpoints. Because Globus Online supports using cloud storage services as these endpoints, this system could potentially be used to support a real-world implementation of Smart-Transfer.

2.2 Cost Minimization

Many researchers have also investigated issues of cost minimization with regard to cloud service usage. While the research described in this section is less likely to be directly useful to the work presented in this thesis or to extensions of this work, it does demonstrate some of the similar yet fundamentally different work being done in other areas of cloud service cost minimization.

Some of this work, like the work presented in this thesis, focuses on dynamic scaling of resources to avoid overpaying. Placek, et al. [63] and Mao, et al. [48] both investigate this issue of dynamic resource scaling. The research of Placek, et al., like ours, is concerned with avoiding overpaying for data storage for institutions with fluctuating requirements. However, this solution differs fundamentally from our presented solution in that it relies on an auction-based marketplace of storage services rather than an efficient data storage and transfer scheme. The work of Mao, et al. is also similar to our work in that it proposes a solution for automatically scaling cloud resources so as to minimize costs while keeping in accordance with a set of already-known performance requirements. However, this research deals with the creation and use of virtual machines to carry out jobs rather than cloud

storage usage. Their solution also uses dynamic scaling in response to real-time monitoring as opposed to using a schedule generated beforehand based on predictions.

CloudCmp [41] is a system which is able to compare and evaluate different cloud providers across a range of metrics to help determine the right choice for a customer. This is similar to our work in that it considers cost and performance levels of different cloud services to help customers save money on cloud costs. However, this system simply provides evaluation to help guide customers in choosing which cloud provider to use, rather than providing a dynamic, time-varying, optimized solution targeted to the specific requirements of the customer.

Perhaps most similar to our work is the work of Tang, et al. [65]. They present Grandet, a storage system which aims to minimize cloud storage costs by making smart choices regarding which storage services to use. As with Smart-Transfer, Grandet relies on predictions of workload patterns in order to determine which storage services could best serve anticipated requests. However, Grandet differs significantly from Smart-Transfer in that its decision-making is primarily focused on choosing which services to use to permanently emplace new data, as well as which services should be used to serve requests for data which has been replicated across multiple storage services. Smart-Transfer, in contrast, focuses on the movement of data between cloud services to smartly serve predicted requests while avoiding overpaying.

Chapter 3

PROBLEM DESCRIPTION

In this chapter, we describe the system model (comprising models for storage services, data access requests, and data transfer actions) and provide examples of the various components of the model. We also describe the preprocessing steps which are potentially necessary to properly format request sequences to be used as input for the Smart-Transfer algorithm. Finally, we present the formal definition of the cost-minimization data storage problem studied in this thesis. A list of notations used in this chapter (Table 3.1) is presented at the end of the chapter.

3.1 System Model

We consider the cloud back-end of a system with large data sets— such as IoT or online video streaming — which can utilize different cloud storage services in a holistic way where one or more data sets can be continually moved between and stored in different storage services.

In the system, there are M different cloud storage services. Each service i has a tuple C_i to model its cost (i.e., bill charge) on data operations. $\mathrm{C}_i = \langle \alpha_i, \beta_i, \gamma_i \rangle$, where α_i is the storage cost per data and time unit, and β_i and γ_i are the outgoing and incoming data-transfer costs per data unit, respectively.

Let $\hat{\alpha}$ and $\check{\alpha}$ be the storage costs for the highest- and the lowest-end services available in the system. The storage cost for service i is defined as:

$$\alpha_i = \frac{(i-1)(\hat{\alpha} - \check{\alpha})}{(M-1)} + \check{\alpha}. \tag{3.1}$$

In our model, the performance level of a storage service is directly reflected by its storage

cost, i.e., the higher the cost is, the better the performance is. This also matches the pricing and performance settings of cloud storage services offered by AWS (e.g., standard and glacier simple storage service classes) or Azure (e.g., hot and cold blobs). For simplicity, we use the storage cost to represent the performance level of a storage service throughout this thesis.

When data is transferred from service i to j, the associated data-transfer delay is defined as:

$$\tau_{i \to j} = \frac{\phi}{v_{i \to j}}, \tag{3.2}$$

where ϕ is the size of target data and $v_{i \to j}$ is the data-transfer speed from service i to j. Such transfer latency requires that data must be transferred to a necessary storage service ahead of the actual data-access time to avoid unnecessary performance violation.

A user may generate a data-access request for some data in the system. Such a request is modeled as $\mathrm{R}_j = \langle r_j, T_j, \lambda_j \rangle$, where r_j is the minimum-level storage service to store the data without negatively impacting user experience, T_j is the access starting time, and λ_j is the access duration. Smart-Transfer requires predictions of such data access requests in order to function. For a certain period of time, multiple data-access requests for the same data can be predicted as a sequence of N requests incrementally ordered by the access starting time. The generation of such predictions is outside the scope of this thesis. However, research has been done regarding the generation of such predictions [21, 64, 65], and a scheme similar to one of these could potentially be adapted for use by Smart-Transfer.

To satisfy one or more requests, a data-transfer action $\mathrm{X}_k = \langle u_k, v_k, T'_k, \lambda'_k \rangle$ can be made to transfer data from source storage service u_k to destination service v_k starting from time T'_k, after which the data will remain at v_k for a duration indicated by λ'_k.

3.2 Preprocessing

For the evaluations performed in this research, it is assumed that the request sequence used as input consists of a list of non-overlapping requests for a single set of data with no points in time between the first and last requests during which there is no request for the data.

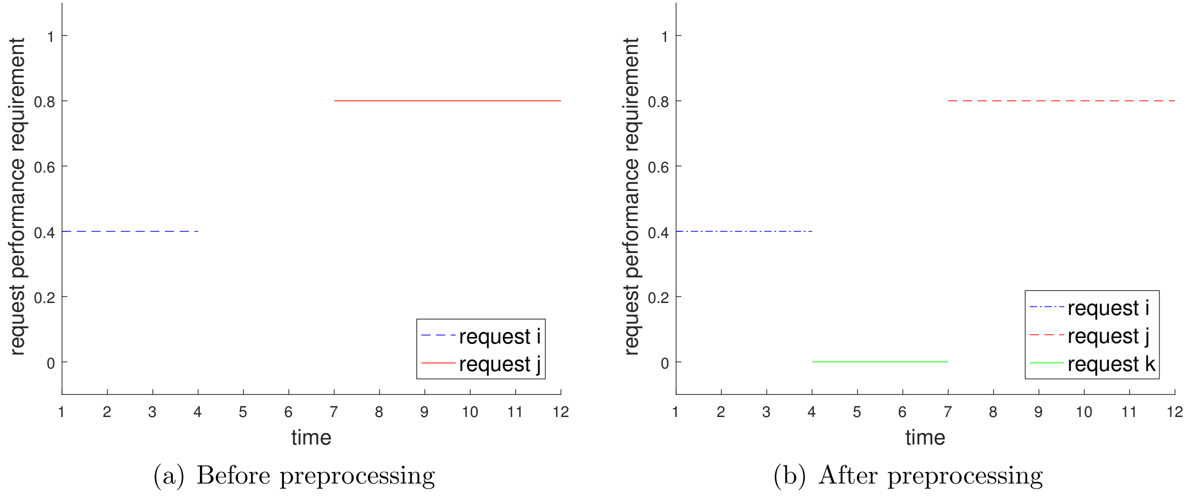
Furthermore, to ensure the proper functioning of the Smart-Transfer algorithm, it is assumed that no requests in the request sequence are of a shorter duration than the transfer delay for the requested data. In practice, actual requests may not conform to these specifications, and thus predictions of those requests may also not conform to these specifications. However, given a request sequence which does not meet these requirements, a set of preprocessing operations can be performed to bring the request sequence in line with these requirements.

To ensure that the input meets the requirement that a request sequence only consist of data access requests for a specific set of data, requests for different data sets can be separated from each other into separate request sequences. The Smart-Transfer solution can then be applied to each of the resulting request sequences. This allows each set of data to have its own transfer sequence generated for it based on the demand for that specific data set.

To ensure that there is an active request associated with all points in time during the first and last request, for any stretch of time during which no request exists in the request sequence, a new request can be created for that time period which requests the lowest possible performance requirement. Specifically, for any two points in time t_i and t_j such that no requests in the request sequence have a request period which overlaps with the period between t_i and t_j, this time period can be covered by a new request $\mathrm{R}_k = \langle t_i, r_0, t_j - t_i \rangle$. Figure 3.1 shows an example of this.

Overlapping requests for the same set of data can be reformed into non-overlapping requests while still serving all requests. The simplest case is where one request is completely overlapped by another request for an equal or higher performance requirement. In such a case, the covered request can be removed from the request list, as it is assumed that any storage service which is able to serve the request for the higher performance requirement is also able to serve the request for the lower performance requirement. Specifically, for R_i and R_j such that $r_i \geqslant r_j$, $T_i \leqslant T_j$, and $T_i + \lambda_i \geqslant T_j + \lambda_j$, R_j can be removed from the request list. This is demonstrated in Figure 3.2.

If one request is completely overlapped by another request for a lower performance requirement, then the portions of the lower-requirement request which do not overlap with



(a) Before preprocessing

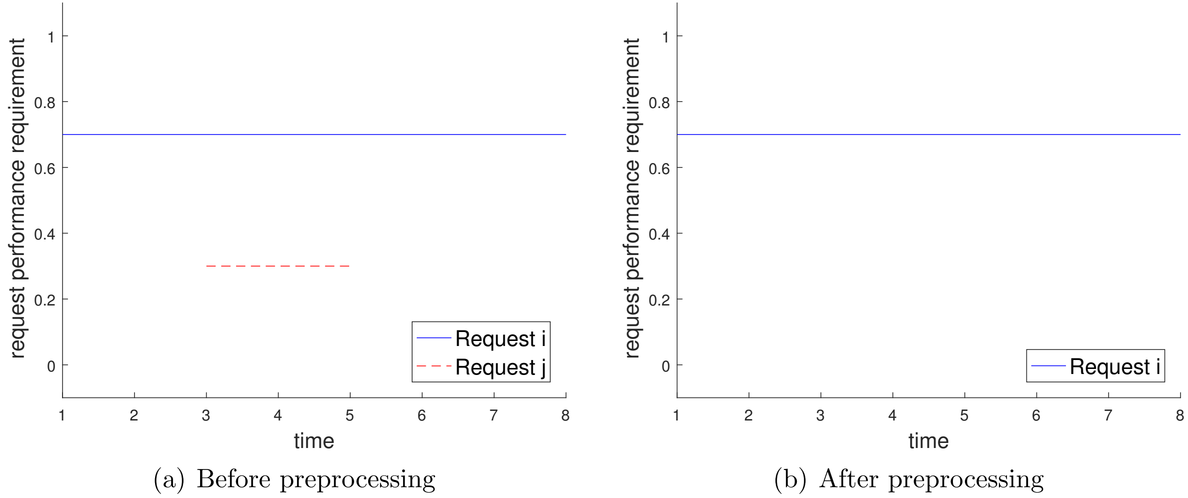
(b) After preprocessing

Figure 3.1: Result of preprocessing to fill in request gaps. For any time window for which no request originally exists, a new request is created for that time window with a performance requirement of 0 (i.e., it can be fulfilled by any storage service).

the higher-requirement request can be split off into new requests on either side of the higher-requirement request. Specifically, for R_i and R_j such that $r_i < r_j$, $T_i < T_j$, and $T_i + \lambda_i > T_j + \lambda_j$, R_i can be replaced with two new requests $\mathrm{R}'_i = \langle r_i, T_i, T_j - T_i \rangle$ and $\mathrm{R}''_i = \langle r_i, T_j + \lambda_j, \lambda''_i \rangle$ where $\lambda''_i = T_i + \lambda_i - T_j - \lambda_j$. This is shown in Figure 3.3.

If two requests partially overlap with each other and request the same performance requirement, then these two requests can be combined into a single request with a start time equal to the start time of the first request and an end time equal to the end time of the second request. Specifically, for R_i and R_j such that $r_i = r_j$ and $T_i + \lambda_i \geqslant T_j$, R_i and R_j can be combined into a new request $\mathrm{R}'_i = \langle r_i, T_i, \lambda'_i \rangle$ where $\lambda'_i = \lambda_j + T_j - T_i$.

If two partially overlapping requests do not specify the same performance requirement, then the overlapping section of the lower-requirement request can be removed, leaving only the higher-requirement request. Specifically, there are two possible cases: either the earlier request has a lower performance requirement than the later request, or else the earlier request has a higher performance requirement than the later request. For R_i and R_j such that $r_i < r_j$ and $T_i + \lambda_i \geqslant T_j$, R_i can be altered such that $\mathrm{R}_i = \langle r_i, T_i, T_j - T_i \rangle$. For R_i and R_j such



(a) Before preprocessing

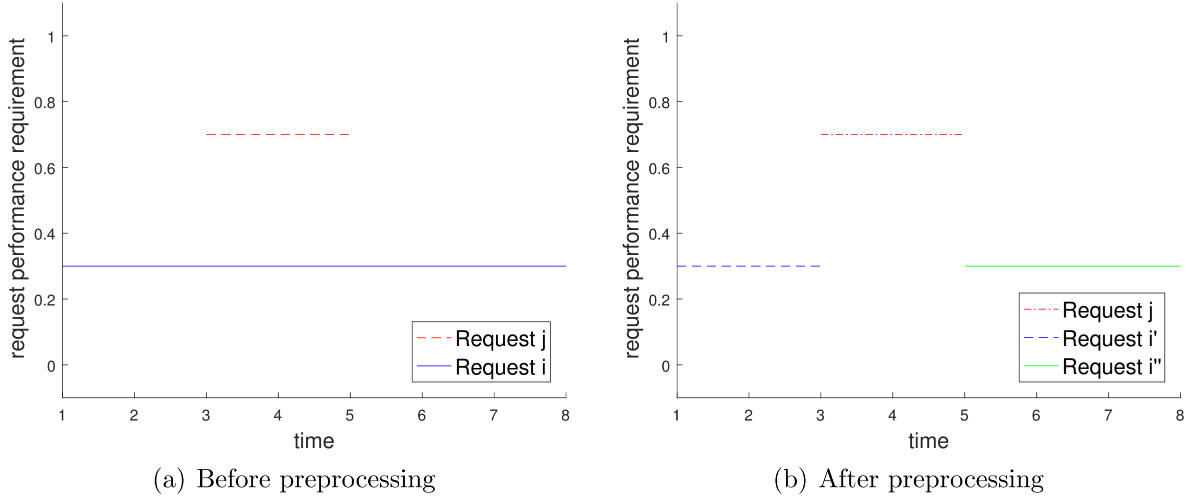
(b) After preprocessing

Figure 3.2: Result of preprocessing to remove redundant requests. In this case, request j is fully covered by request i (which has a higher performance requirement), and so can be removed.

that $r_i > r_j$ and $T_i + \lambda_i \geqslant T_j$, R_j can be altered such that $\mathrm{R}_j = \langle r_j, T_i + \lambda_i, \lambda_j' \rangle$ where $\lambda_j' = T_j + \lambda_j - T_i - \lambda_i$. The second case is demonstrated in Figure 3.4.

The Smart-Transfer algorithm as presented is unable to properly handle requests with durations shorter than the transfer delay for the requested data. To mitigate this issue, such short requests can either be extended or combined with the request either immediately preceding or immediately following. This step must be performed after it is verified that no requests overlap and that all points in time for the time period in question are covered by a request in the request sequence. How to handle a given short request depends on the configuration of the requests on either side of it. The various possibilities are explained below. The goal in all of the following cases is to satisfy all requested performance requirements as well as the restriction that no request be shorter than the transfer delay, while also minimizing the total requested performance requirements. Thus, for example, if two different requests could possibly be extended to take over for some third request which is too short, the one with the lower performance requirement will be the one chosen for extension.

If both the request which immediately precedes the short request and the request which



(a) Before preprocessing

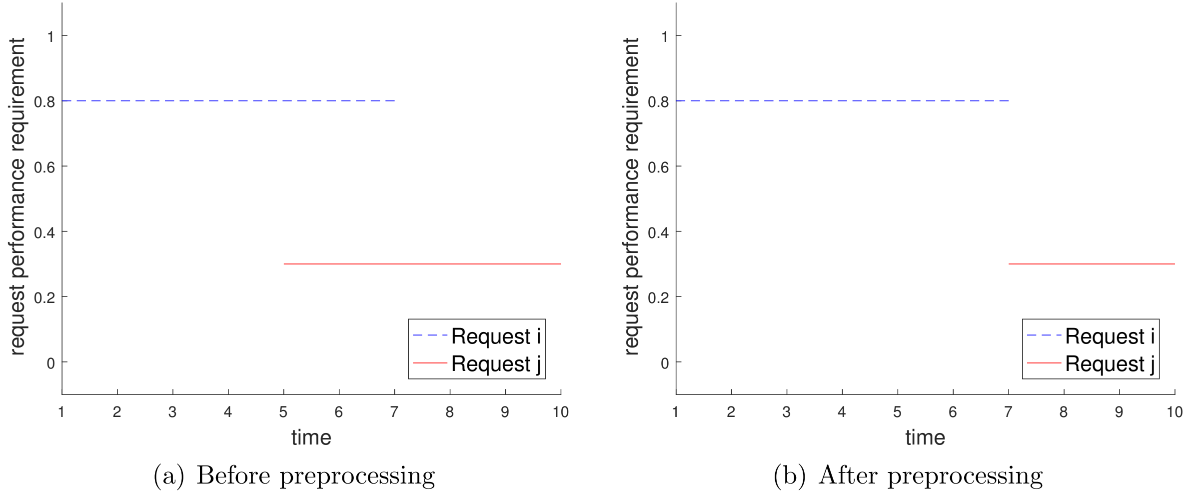
(b) After preprocessing

Figure 3.3: Result of preprocessing to split up a request which is partially covered by a smaller request with a higher performance requirement.

immediately follows the short request are for a higher performance requirement than the short request, then the short request can be taken over by whichever of the two flanking requests has the lower performance requirement. Specifically, for R_i, R_j, and R_k such that $r_i > r_j$, $r_k > r_j$, $T_i < T_j < T_k$, and $\lambda_j < \tau$, R_j will be removed from the request sequence and one of the following two operations will be performed: if $r_i < r_k$, then R_i will be altered such that $\mathrm{R}_i = \langle r_i, T_i, \lambda_i + \lambda_j \rangle$ (Figure 3.5); or, if $r_i > r_k$, then R_k will be altered such that $\mathrm{R}_k = \langle r_k, T_j, \lambda_k + \lambda_j \rangle$.

If, on the other hand, both neighboring requests have a lower performance requirement than the short request, then the short request can be extended towards whichever neighboring request has a higher performance requirement. Specifically, for R_i, R_j, and R_k such that $r_i < r_j$, $r_k < r_j$, $T_i < T_j < T_k$, and $\lambda_j < \tau$, either $r_i > r_k$, in which case R_j will be extended such that $\mathrm{R}_j = \langle r_j, T_j - \tau + \lambda_j, T_j - \tau + 2\lambda_j \rangle$, or else $r_i \leqslant r_k$ in which case R_j will be extended such that $\mathrm{R}_j = \langle r_j, T_j, \tau \rangle$ (Figure 3.6).

If the short request has a performance requirement which is between the performance requirements of the requests on either side of it, then either the short request will be absorbed

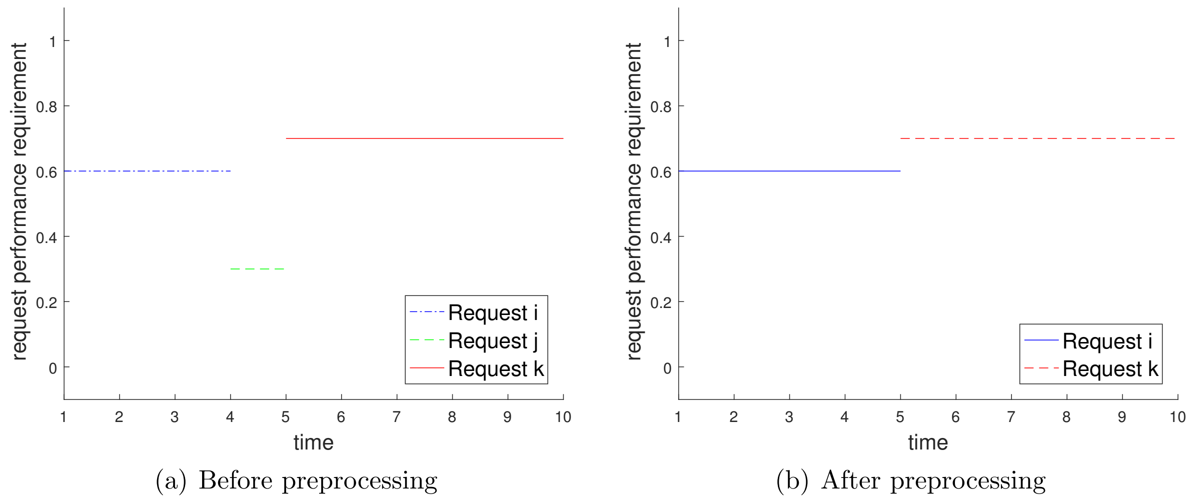


(a) Before preprocessing

(b) After preprocessing

Figure 3.4: Result of preprocessing to shorten a request which is partially overlapped by a request with a higher performance requirement.

by one of its neighbors or else it will extend into the time of the other neighboring request, depending on which neighboring request has a performance requirement which is closer to that of the short request. Specifically, either the preceding request has a lower performance requirement than the short request and the following request has a higher requirement, or vice versa. In the first case, for R_i, R_j, and R_k such that $T_i < T_j < T_k$, $\lambda_j < \tau$, and $r_i < r_j < r_k$, either $r_j - r_i < r_k - r_j$ in which case R_j will be extended such that $\mathrm{R}_j = \langle r_j, T_j - \tau + \lambda_j, T_j - \tau + 2\lambda_j \rangle$ (Figure 3.7), or else $r_j - r_i \geqslant r_k - r_j$ in which case R_j can be removed from the request sequence and R_k can be extended such that $\mathrm{R}_k = \langle r_k, T_j, \lambda_j + \lambda_k \rangle$. In the second case, for R_i, R_j, and R_k such that $T_i < T_j < T_k$, $\lambda_j < \tau$, and $r_i > r_j > r_k$, either $r_j - r_k < r_i - r_j$ in which case R_j will be extended such that $\mathrm{R}_j = \langle r_j, T_j, \tau \rangle$, or else $r_j - r_k \geqslant r_i - r_j$ in which case R_j can be removed from the request sequence and R_i can be extended such that $\mathrm{R}_i = \langle r_i, T_i, \lambda_i + \lambda_j \rangle$.

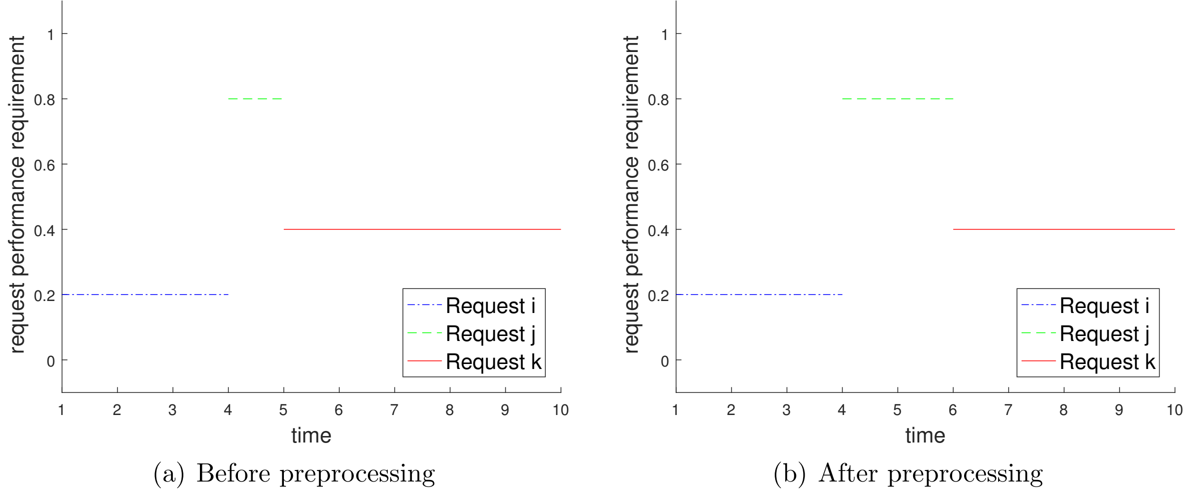


(a) Before preprocessing (b) After preprocessing

Figure 3.5: Result of preprocessing to remove a request which is shorter than the data transfer delay and which is both preceded and followed by requests with higher performance requirements. In this case, the work of that request is taken over by the request which immediately precedes the short request.

3.3 Problem Statement

Formally, the problem studied in this thesis can be described as follows. Notations used in this thesis are summarized in Table 3.1.



(a) Before preprocessing

(b) After preprocessing

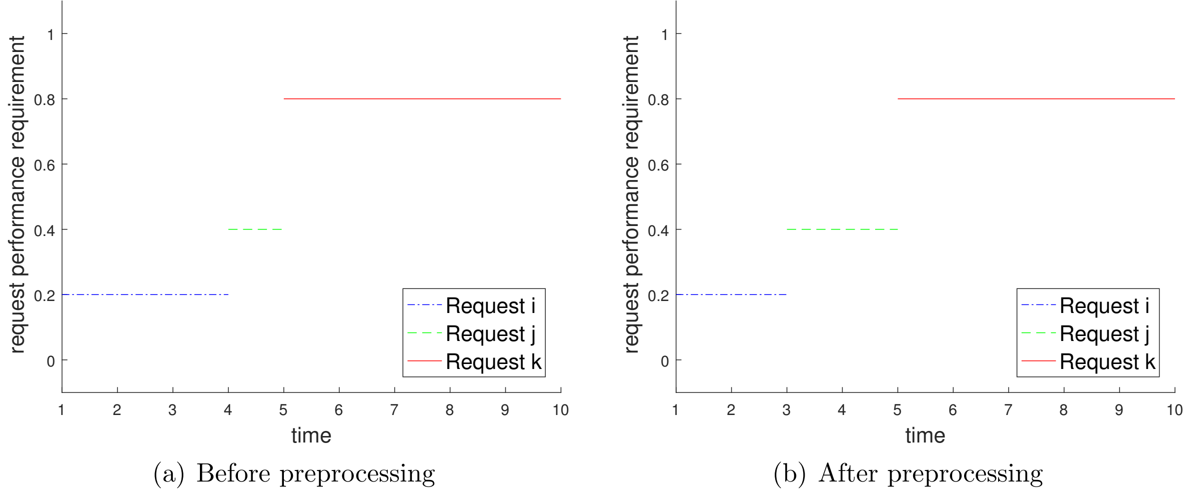
Figure 3.6: Result of preprocessing to extend a request which is shorter than the data transfer delay and which is both preceded and followed by requests with lower performance requirements. In this case, $\tau = 2$.

Objective:

- $\min \sum_{\mathrm{X}_k \in \mathrm{X}} (\beta_{u_k} + \gamma_{v_k} + \alpha_{v_k} \lambda_k^{'}) \cdot \phi$

Given:

- data size: ϕ
- default data storage service: b
- Data-access request sequence: $\mathrm{R} : \langle \mathrm{R}_1, \mathrm{R}_2, \ldots, \mathrm{R}_n \rangle$
- For each service i:
  - unit data-storage cost: α_i
  - unit data-transfer cost: β_i and γ_i
  - data-transfer speed to service j: $v_{i \to j}$



(a) Before preprocessing

(b) After preprocessing

Figure 3.7: Result of preprocessing to extend a request which is shorter than the data transfer delay and which is preceded by a request with a lower performance requirement and followed by a request with a higher performance requirement. In this case, $\tau = 2$.

Subject to:

- *Performance Requirement:*

  $\forall \mathrm{R}_j \in \mathrm{R}$, $\exists \mathrm{X}_k \in \mathrm{X}$:

  - $\alpha_{v_k} \geqslant \alpha_{r_j}$
  - $T'_k + \frac{\phi}{v_{u_k \to v_k}} \leqslant T_j$
  - $T'_k + \frac{\phi}{v_{u_k \to v_k}} + \lambda'_k \geqslant T_j + \lambda_j$

Output:

- Data-transfer action sequence: $\mathrm{X} : \langle \mathrm{X}_1, \mathrm{X}_2, \ldots, \mathrm{X}_k \rangle$

In this problem, there are time-series variables and both linear and non-linear constraints, which makes solving this problem non-trivial. By analyzing this problem, we can observe that this is a classic Dynamic Programming problem, where the data storage location (*state*) may change after each transfer action (*decision*), each action will yield certain costs on data

transfer and storage (*value function*), and the goal is to minimize the total cost (*optimization*).

Table 3.1: Summary of Notations

notation	meaning
C_i	cost tuple of storage service i
α_i β_i γ_i	storage cost per data and time unit of C_i outgoing transfer cost of C_i incoming transfer cost of C_i
R_j	data-access request j
r_j T_j λ_j	the minimum storage service capable of fulfilling R_j start time of R_j data-access duration required by R_j
X_k	transfer action k
u_k v_k T'_k λ'_k	source service of X_k destination service of X_k data-transfer start time of X_k data-storage duration after executing transfer action of X_k
$\check{\alpha}$	storage cost of the lowest-level storage service
$\hat{\alpha}$	storage cost of the highest-level storage service
$\tau_{i\to j}$	data-transfer delay between services i and j
ϕ	data size
$v_{i\to j}$	data-transfer speed from service i to j

Chapter 4
DESIGN OF SMART-TRANSFER SCHEME

In this chapter, we present the details of the proposed Smart-Transfer scheme. We present two primary algorithms (Algorithms 1 and 2) along with accompanying explanations. These algorithms represent the main, single-service version of the Smart-Transfer solution. We also present descriptions of two variants to Smart-Transfer - the double-service and triple-service solutions. The modifications to Algorithms 1 and 2 which are required for the double-service variation are demonstrated, along with explanations of both variations.

4.1 General Dynamic Programming Solution

The primary version of Smart-Transfer utilizes an algorithm to compute a cost-minimized sequence of transfers which will fulfill all of the requests in the request sequence input using a given set of input services. This algorithm is split into two separate sections which are shown in Algorithms 1 and 2.

Algorithm 1 begins by computing and storing the costs that would be incurred by fulfilling the first request at each service capable of doing so (line 4). Each of these costs corresponds to a single possible transfer action choice (or candidate) and is stored as a tuple containing the service the data were at before this choice (source), the service used for this choice (destination), and the cost computed for this choice (lines 3 $\sim$ 11). This process is then repeated for every following request in the request sequence (lines 12 $\sim$ 29). Specifically, for all services capable of fulfilling the current request (lines 15 $\sim$ 16), the algorithm examines all of the candidate actions chosen for the previous request (lines 20 $\sim$ 21), and new transfer action choices can be made based on the accumulative storage and transfer cost (lines 20 $\sim$ 28). Once this process is completed, the output is a list of n candidate sets, each of which may contain up to m choices. Each of these sets of choices represents the cheapest data-

transfer actions capable of fulfilling the associated request by using the indicated service. For example, if request i can only be satisfied by two services, then candidate set i would contain two choices, each of which contains the information about the cheapest transfer action to the corresponding service. A flowchart of this algorithm is depicted in Figure 4.1.

Once this list of tuple sets has been built, the final transfer sequence can be built using Algorithm 2. This is done by beginning with the final candidate set in the list and finding the candidate with the lowest cost in the set (lines $6 \sim 13$). This candidate's cost is the total cost of fulfilling the entire request sequence. Once this has been found, the entire list of cheapest candidate sets can be traced back (lines $14 \sim 20$). The result of this process is a sequence of n services that can be used to fulfill the n requests. To build the final output list of transfer action tuples, this list is condensed such that each subsequence of one or more entries of the same service is used to create a single transfer action tuple which covers all of the requests satisfied by entries in the subsequence (lines $21 \sim 31$). Once this process is completed, the result is a list of all of the transfers required to achieve the lowest-cost solution found by the algorithm. A flowchart for this algorithm is depicted in Figure 4.2.

Assuming that the number of requests (N) is much higher than the number of available storage services in the system (M),

The time complexities for Algorithms 1 and 2 are $\mathcal{O}(NM^2)$ and $\mathcal{O}(N)$, respectively, and the overall solution has a time complexity of $\mathcal{O}(NM^2)$, where N is the number of requests in the request sequence and M is the number of available storage services. While this time complexity may appear to be unreasonably large, the limited number of storage services available to users puts a hard limit on the size of M, whereas the size of N has no upper bound. Furthermore, as will be demonstrated later in Chapter 5, the performance increase gained by adding additional available storage services decreases sharply beyond a small value of M. Thus it is unlikely that M will be a large number, meaning that the time complexity will often be dominated by the N term.

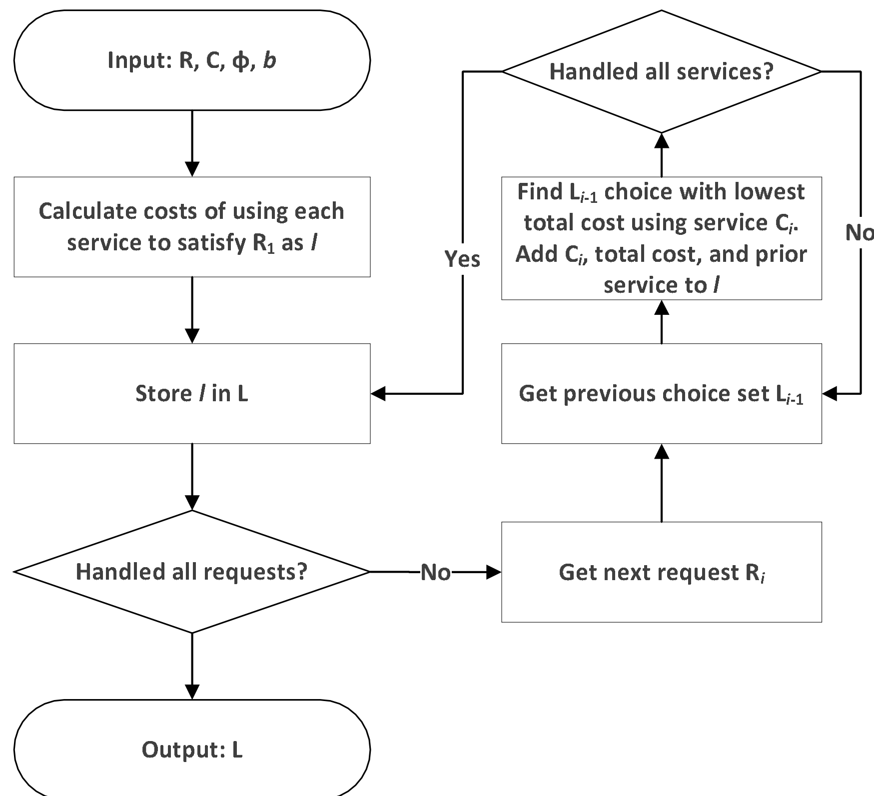


Figure 4.1: Flowchart depicting Algorithm 1.

4.2 Variations

The primary version of Smart-Transfer only allows for the requested data to be stored at a single storage service at a time. However, there are instances where it is the case that temporarily storing the data at multiple storage services can achieve a lower overall cost. To compare this solution with the one that is closer to optimal, two modified versions of Smart-Transfer are also considered. One of these variations, referred to as the "double-service solution," considers storing data in up to two services simultaneously. The modifications to Algorithms 1 and 2 required for this variation are shown in Algorithms 3, 4, and 5. This solution also uses a modified version of the candidate choice tuples used in the original solution. These tuples consist of the primary source service, the secondary source service,

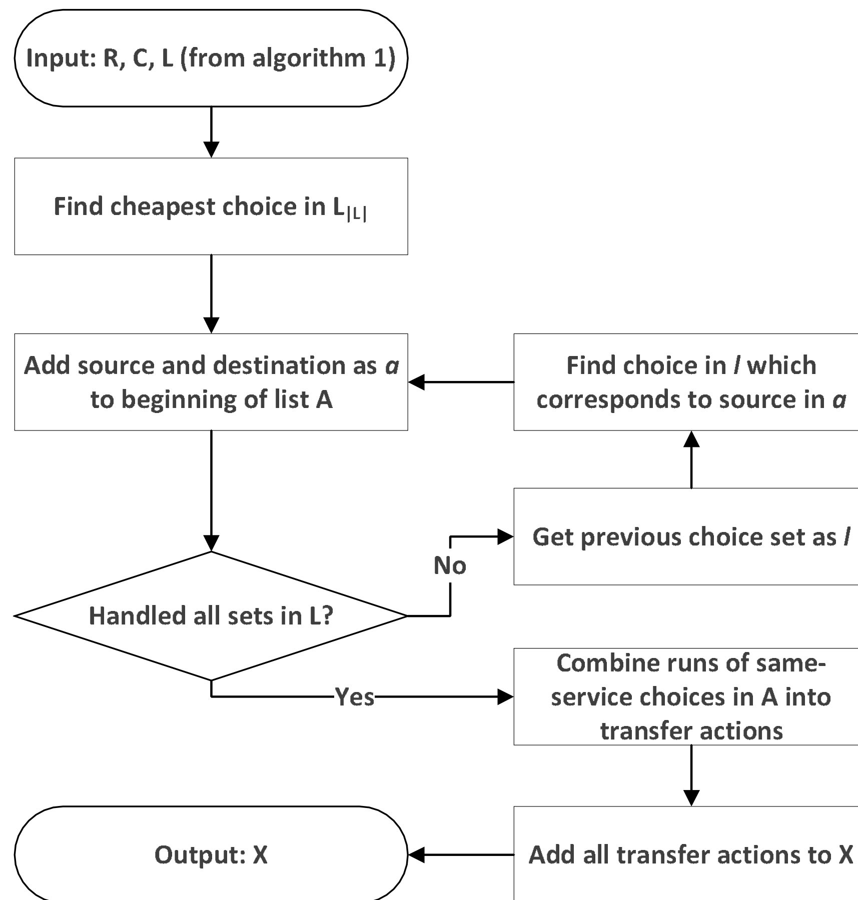


Figure 4.2: Flowchart depicting Algorithm 2.

the primary destination service, the secondary destination service, and the cost computed for the corresponding candidate choice. The output of Algorithm 2 is also likewise modified such that each transfer action in the list includes a secondary source service and a secondary destination service.

The second variation, referred to as the "triple-service solution," considers storing data in up to three services simultaneously. This variation is achieved by further extending the Smart-Transfer algorithms in a similar manner to the extensions required for the double-service solution. The performance of these two variations are also explored and evaluated via simulations alongside the original version of Smart-Transfer, and the obtained results

show that in some cases using the double-service solution yields better results than the original version, the triple-service solution is not demonstrated to ever improve significantly over the double-service solution.

Algorithm 1 The data-transfer candidate list construction algorithm

Input:

- $\langle \mathrm{R}_1, \ldots, \mathrm{R}_n \rangle$: list of data-access requests sorted in the ascending order of starting time
- $\langle \mathrm{C}_1, \ldots, \mathrm{C}_m \rangle$: list of available storage services in the system
- ϕ: data size
- b: initial data storage service

Output:

- L: list of data-transfer action list

```
 1: L ← ∅                                        /* list of data-transfer action candidates */
 2: l ← ∅                                        /* list of data-transfer candidate tuples */
 3: for i = 1 to m do
 4:    if α_i ⩾ α_{r_1} then
 5:       src ← b
 6:       dst ← i
 7:       cost ← λ_1 α_i φ
 8:       if src ≠ dst then
 9:          cost ← cost + (β_b + γ_i + τ_{b→i} α_i)φ
10:       l[i] ← ⟨src, dst, cost⟩
11: L ← L ∪ l                                    /* the array of candidates is appended */
12: for j = 2 to n do
13:    l ← ∅
14:    p ← L[|L|]                                /* obtain the previous array of candidates */
15:    for i = 1 to m do
16:       if α_i ⩾ α_{r_j} then
17:          src ← ∅
18:          dst ← i
19:          cost ← +∞
20:          for k = 1 to m do
21:             if p[k] ≠ ∅ then
22:                Δ ← λ_j α_i φ + p[k].cost
23:                if k ≠ i then
24:                   Δ ← Δ + (β_k + γ_i + τ_{k→i} α_i)φ
25:                if Δ < cost then
26:                   cost ← Δ
27:                   src ← k
28:          l[i] ← ⟨src, dst, cost⟩
29:    L ← L ∪ l                                 /* the array of candidates is appended */
30: return L
```

Algorithm 2 The data-transfer sequence construction algorithm

Input:

- $\langle \mathrm{R}_1, \ldots, \mathrm{R}_n \rangle$: list of data-access requests sorted in the ascending order of starting time
- $\langle \mathrm{C}_1, \ldots, \mathrm{C}_m \rangle$: list of available storage services in the system
- L: the output list from Algorithm 1

Output:

- X: list of data-transfer actions

1: $A \leftarrow \emptyset$ */* list of transfer source-destination pairs*/*
2: $l \leftarrow L[|L|]$
3: $L \leftarrow L - l$
4: $cost^* \leftarrow +\infty$
5: $src \leftarrow \emptyset$
6: **for** $i = 1$ to $|l|$ **do**
7:   **if** $l[i] \neq \emptyset$ **then**
8:     **if** $l[i].cost < cost^*$ **then**
9:       $cost^* \leftarrow l[i].cost$
10:       $dst \leftarrow l[i].dst$
11:       $src \leftarrow l[i].src$
12: $a \leftarrow \langle src, dst \rangle$
13: $A \leftarrow A \cup a$
14: **for** $i = n - 1$ to 1 **do**
15:   $l \leftarrow L[|L|]$
16:   $L \leftarrow L - l$
17:   $dst \leftarrow src$
18:   $src \leftarrow l[dst].src$
19:   $a \leftarrow \langle src, dst \rangle$
20:   $A \leftarrow A \cup a$
21: $\mathrm{X} \leftarrow \emptyset$
22: $last \leftarrow 1$
23: $src \leftarrow b$
24: **for** $i = 2$ to n **do**
25:   **if** $A[i].dst \neq A[i].src$ **then**
26:     $dst \leftarrow A[i].src$
27:     $x \leftarrow \langle src, dst, T_{last} - \tau_{src \to dst}, T_i - T_{last} \rangle$
28:     $last \leftarrow i$
29:     $src \leftarrow dst$
30:     $\mathrm{X} \leftarrow \mathrm{X} \cup x$
31: $x \leftarrow \langle src, A[n].dst, T_{last}, T_n + \lambda_n - T_{last} \rangle$
32: $\mathrm{X} \leftarrow \mathrm{X} \cup x$
33: **return** X

Algorithm 3 For the double-service variation, these lines replace line 10 in Algorithm 1

1: **for** $j = 0$ to $i - 1$ **do**
2: $\quad$ $secdst \leftarrow j$
3: $\quad$ **if** $j \neq 0$ **then**
4: $\quad\quad$ $cost \leftarrow cost + \lambda_i \alpha_i \phi$
5: $\quad\quad$ **if** $j \neq src$ **then**
6: $\quad\quad\quad$ $cost \leftarrow cost + (\beta_b + \gamma_j + \tau_{b \to j} \alpha_j)\phi$
7: $\quad$ $l \leftarrow l \cup \langle src, 0, dst, secdst, cost \rangle$

Algorithm 4 For the double-service variation, these lines replace line 16 through 28 in Algorithm 1

1: **for** $k = 0$ to $i - 1$ **do**
2: $\quad$ **if** $\alpha_i \geqslant \alpha_{r_j}$ **then**
3: $\quad\quad$ $src \leftarrow \emptyset$
4: $\quad\quad$ $secsrc \leftarrow \emptyset$
5: $\quad\quad$ $dst \leftarrow i$
6: $\quad\quad$ $secdst \leftarrow k$
7: $\quad\quad$ $cost \leftarrow +\infty$
8: $\quad\quad$ **for** $c = 1$ to $|p|$ **do**
9: $\quad\quad\quad$ **if** $p[c] \neq \emptyset$ **then**
10: $\quad\quad\quad\quad$ $\Delta \leftarrow \lambda_j \alpha_i \phi + p[c].cost$
11: $\quad\quad\quad\quad$ **if** $p[c].dst \neq i$ **AND** $p[c].secdst \neq i$ **then**
12: $\quad\quad\quad\quad\quad$ $\Delta \leftarrow \Delta + (\beta_{p[c].dst} + \gamma_i + \tau_{p[c].dst \to i} \alpha_i)\phi$
13: $\quad\quad\quad\quad$ **if** $k \neq 0$ **then**
14: $\quad\quad\quad\quad\quad$ $\Delta \leftarrow \Delta + \lambda_j \alpha_k \Phi + p[c].cost$
15: $\quad\quad\quad\quad\quad$ **if** $p[c].dst \neq k$ **AND** $p[c].secdst \neq k$ **then**
16: $\quad\quad\quad\quad\quad\quad$ $\Delta \leftarrow \Delta + (\beta_p[c].dst + \gamma_k + \tau_{p[c].dst \to k} \alpha_k)\phi$
17: $\quad\quad\quad\quad$ **if** $\Delta < cost$ **then**
18: $\quad\quad\quad\quad\quad$ $cost \leftarrow \Delta$
19: $\quad\quad\quad\quad\quad$ $src \leftarrow p[c].dst$
20: $\quad\quad\quad\quad\quad$ $secsrc \leftarrow p[c].secdst$
21: $\quad\quad$ $l \leftarrow l \cup \langle src, secsrc, dst, secdst, cost \rangle$

Algorithm 5 For the double-service variation, these lines replace lines 10 through 33 in Algorithm 2

1: **for** $i = 1$ to $|l|$ **do**
2:   **if** $l[i] \neq \emptyset$ **then**
3:     **if** $l[i].cost < cost*$ **then**
4:       $cost* \leftarrow l[i].cost$
5:       $dst \leftarrow l[i].dst$
6:       $secdst \leftarrow l[i].secdst$
7:       $src \leftarrow l[i].src$
8:       $secsrc \leftarrow l[i].secsrc$
9: $a \leftarrow \langle src, secsrc, dst, secdst \rangle$
10: $A \leftarrow A \cup a$
11: **for** $i = n - 1$ to 1 **do**
12:   $l \leftarrow L[|L|]$
13:   $L \leftarrow L - l$
14:   $dst \leftarrow src$
15:   $secdst \leftarrow secsrc$
16:   $prev \leftarrow \emptyset$
17:   **for** $j = 1$ to $|l|$ **do**
18:     **if** $dst = l[j].dst$ **AND** $secdst = l[j].secdst$ **then**
19:       $prev \leftarrow l[j]$
20:       break
21:   $src \leftarrow prev.src$
22:   $secsrc \leftarrow prev.secsrc$
23:   $a \leftarrow \langle src, secsrc, dst, secdst \rangle$
24:   $A \leftarrow A \cup a$
25: $\mathrm{X} \leftarrow \emptyset$
26: $last \leftarrow 1$
27: $src \leftarrow b$
28: $secsrc \leftarrow 0$
29: **for** $i = 2$ to n **do**
30:   **if** $A[i].dst \neq A[i].src$ **then**
31:     $dst \leftarrow A[i].src$
32:     $x \leftarrow \langle src, secsrc, dst, secdst, T_{last} - \tau_{src \rightarrow dst}, T_i - T_{last} \rangle$
33:     $last \leftarrow i$
34:     $src \leftarrow dst$
35:     $secsrc \leftarrow secdst$
36:     $\mathrm{X} \leftarrow \mathrm{X} \cup x$
37: $x \leftarrow \langle src, A[n].dst, T_{last}, T_n + \lambda_n - T_{last} \rangle$
38: $\mathrm{X} \leftarrow \mathrm{X} \cup x$
39: **return** X

Chapter 5

PERFORMANCE EVALUATION

Via extensive simulations, we have evaluated the performance of Smart-Transfer in terms of total cost of data storage and transfer given a request sequence and compared Smart-Transfer with the following schemes.

- No-Transfer: a scheme that simply stores data at one storage service which is able to fulfill each request in the request sequence.

- All-Transfer: a scheme that always uses the cheapest but capable storage service to fulfill each request in the request sequence.

In this chapter, we present the results of these simulations as well as analysis of these results. The figures used in this chapter all show results for multiple versions of the Smart-Transfer scheme. When one, two, or three storage services are utilized to simultaneously store data, the corresponding Smart-Transfer schemes are respectively labeled as "Single," "Double," and "Triple" in the figures.

5.1 Simulation Setup

In the simulations, the proposed Smart-Transfer scheme is evaluated with changes of the following system parameters:

- Average request duration. This parameter determines the duration of a request in units of time. Each request is randomly generated using a Zero-Truncated Poisson distribution with the value of this parameter as the expected value.

- Idle percentage. This is a value between 0 and 1 indicating the percentage of the total simulation time (on average) that should be spent in idle periods (no request). Thus, every time a new request is generated, there is a probability equal to the value of this parameter that the new request will represent an idle period (i.e., the request can be fulfilled by any storage service).

- Ratio of data-transfer cost to data-storage cost. In the simulation, we set the data incoming transfer cost to be 0 which matches the pricing strategy of most cloud service providers, and we only use the ratio between data outgoing transfer cost and storage cost to evaluate the effect of data-transfer cost on the total cost.

- Number of available storage services in system. This parameter determines how many storage services are available for different schemes to choose when making data transfer and storage decisions.

5.2 Simulation Results

We first evaluate Smart-Transfer with various numbers of available storage services in system. Simulation results are plotted in Figure 5.1. The results show that, in terms of overall cost, the single-service version of Smart-Transfer and the double-service variant both performed as well as or better than the No-Transfer and All-Transfer schemes (naive schemes) in all instances. The results also indicate that there exists a range of circumstances under which Smart-Transfer performs significantly better than the naive schemes with the single-service solution achieving a maximum improvement of approximately 35% and the double-service solution achieving a maximum improvement of approximately 41%.

Given that the performance of Smart-Transfer does not increase significantly beyond a handful of available services (as shown in Figure 5.1), in the rest of this chapter we only show the evaluation results of Smart-Transfer when there are 5 total available storage services in system.

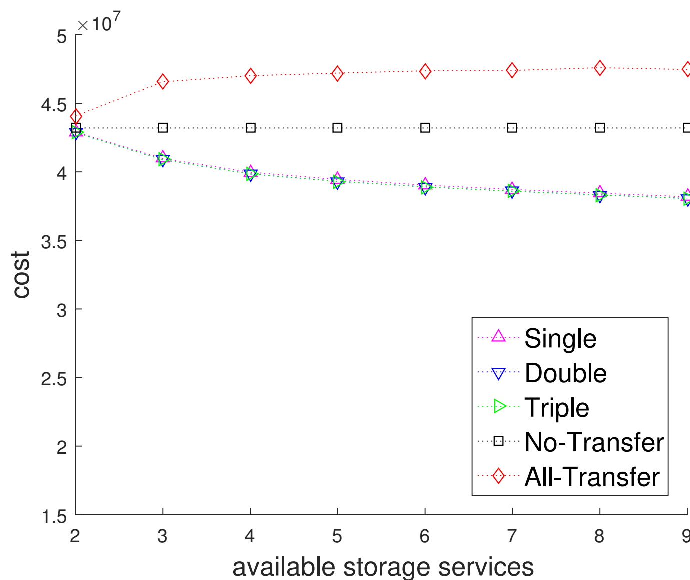
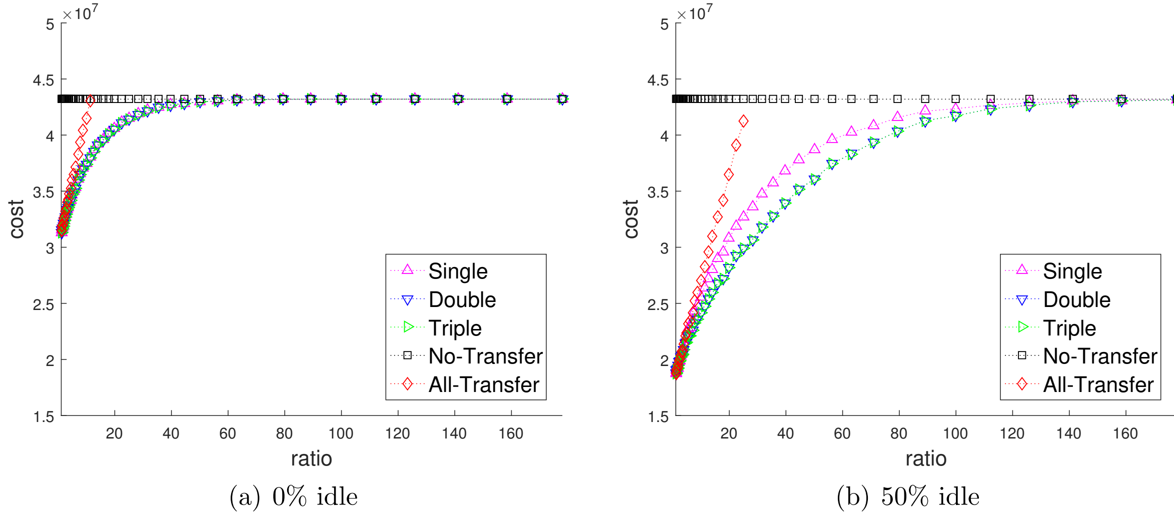


Figure 5.1: Total cost on data storage and transfer under different numbers of total available storage services: the average request duration is 15 time units, the idle percentage is 0, the ratio of data-transfer cost to data-storage cost is 15, and the total simulation time is 43, 200 time units.

Figure 5.2 shows the total cost of each scheme under different ratios of data-transfer cost to data-storage cost, while Figure 5.3 shows the improvement achieved by Smart-Transfer over the two comparison schemes with different ratios. From Figure 5.2, it can be found that the All-Transfer scheme shows a dramatically increased cost trend and bypass other schemes when the ratio becomes larger. This is due to both the unnecessary number of data-transfer activities and the increased cost per data transfer. Therefore, we only show the cost of the All-Transfer scheme when it is lowers than the No-Transfer scheme. This is also applied to the rest of figures when showing the performance of the All-Transfer scheme. Figure 5.3(a) shows that with 0% idle percentage, Smart-Transfer performs best with a transfer-to-storage cost ratio of 12, and Figure 5.3(b) shows that with 50% idle percentage, it performs best with a ratio of 28. These ratio values were used for further simulations.

Figure 5.4 shows the total cost of each scheme under different average request durations. These plots demonstrate how, for very large or very small average request durations, our

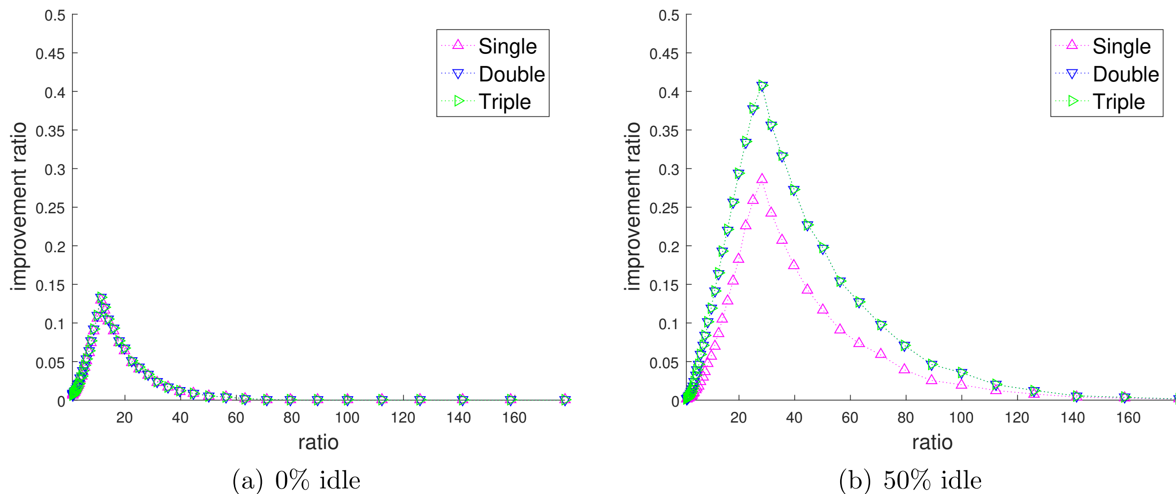


(a) 0% idle

(b) 50% idle

Figure 5.2: Total cost on data storage and transfer under different ratios of data-transfer to data-storage cost: 5 storage services are available, the average request duration is 15, and the total simulation time is 43, 200 time units. The cost for the All-Transfer solution is only displayed when it is lower than that of the No-Transfers solution.

solutions costs approach or equal that of the All-Transfer and No-Transfer solutions, respectively. This is due to the fact that, for very long request durations (e.g., 60 time units), the storage cost of the chosen service dominates the transfer cost and so the optimal choice for any such request is to simply use the service which can most cheaply fulfill that request (which is what the All-Transfer solution does). Conversely, for very short request durations (e.g., 2 time units), the storage costs associated with each individual request are potentially dominated by the cost of frequent transfers. In such cases, the optimal solution is to simply fulfill all requirements using a single, high-end service (which is what the No-Transfer solution does). However, in between these extreme values our solutions show noticeable improvements over the naive solutions, as shown in Figure 5.5. Specifically, when using an idle percentage of 0%, the single-service solution achieved a maximum improvement of 12% while the double-service solution achieved a maximum improvement of 13%. When the idle percentage is 50%, the single-service solution achieved a maximum improvement of 29% while the double-service solution achieved a maximum improvement of 41%. This



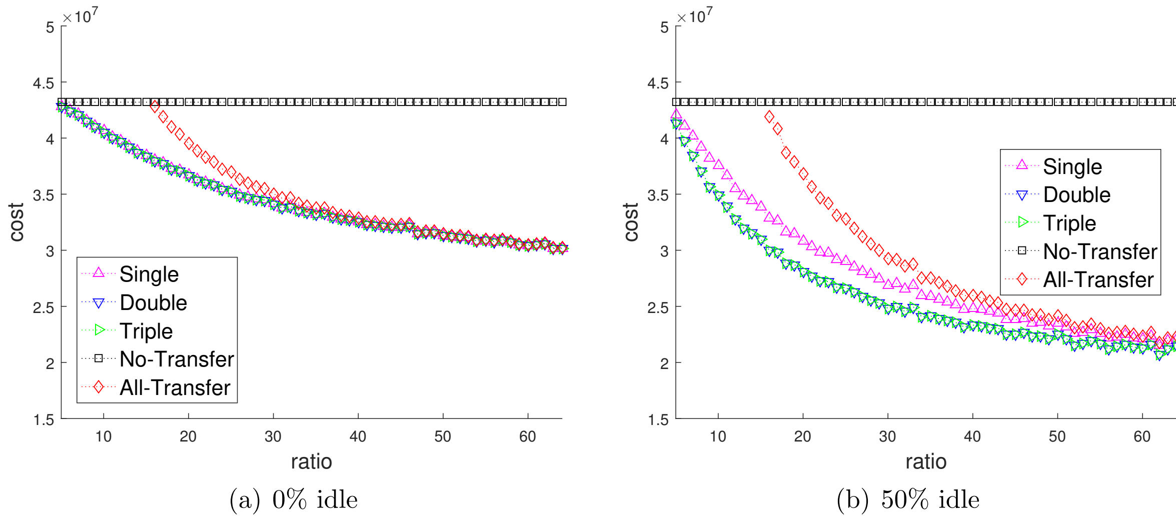
(a) 0% idle (b) 50% idle

Figure 5.3: Improvement ratio of Smart-Transfer to the All-Transfer or No-Transfer scheme that yields a lower cost: 5 storage services are available, the average request duration is 15 time unites, and the total simulation time is $43,200$ time units.

improvement over the 0% idle percentage simulations is because having relatively frequent idle periods gives Smart-Transfer more opportunities to make smart choices which improve over the naive solutions (e.g., leaving the data at a higher-end service during a short idle period as opposed to transferring the data down and then back up).

Figures 5.4 and 5.5 also demonstrate the fact that the improvement gained by using the double-service solution rather than the single-service solution increases with a higher idle percentage. This is due to the fact that the double-service solution performs better than the single-service solution primarily in instances where a low-requirement request is followed by a relatively short higher-requirement request, which is then followed by another request at the same low requirement. In such cases, the double-service solution can leave the data at the low-level service for the duration of the higher-requirement request, then simply terminate the storage at the higher service at the end of the high-requirement request, thereby saving the cost of a transfer. The frequency of such instances increases as the idle percentage increases.

Smart-Transfer performs even better when the requests are in certain patterns. Fig-



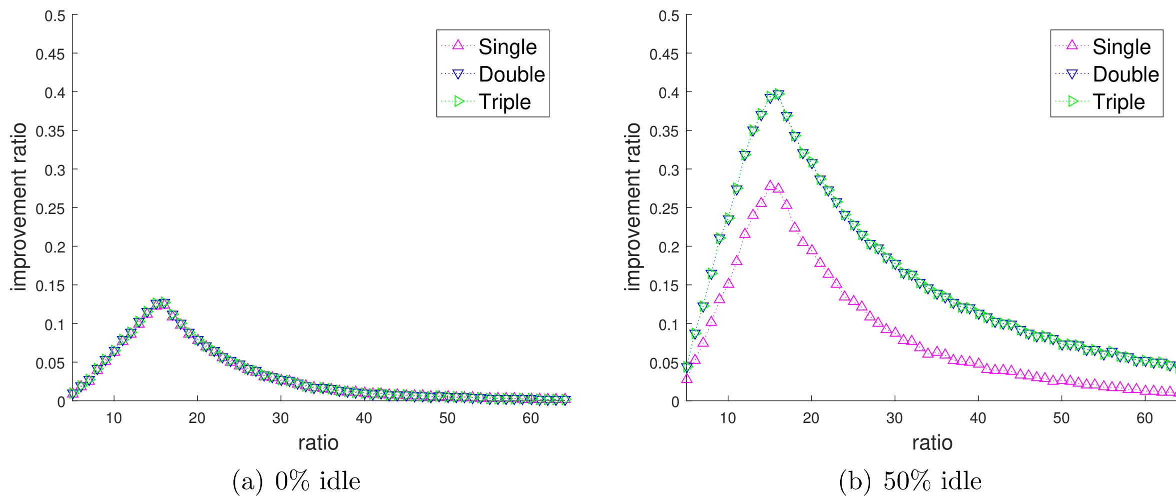
(a) 0% idle

(b) 50% idle

Figure 5.4: Total cost on data storage and transfer under different average request durations: 5 storage services are available, the ratio of data-transfer cost to data-storage cost is 12 for 5.4(a) and 28 for 5.4(b), and the total simulation time is 43,200 time units.

ure 5.6(a) shows the solution costs of the single- and double-service solutions along with the two naive solutions using a request sequence which follows a simple 24-hour cycle of increasing (in daytime) and decreasing (at night) loads. Figure 5.6(b) shows that, while the double-service solution does not perform as well in this case compared to using random requests, the less computationally-expensive single-service solution performs as well as the double-service solution and much better than in the tests using random requests, with both the single- and double-service solutions achieving an improvement of 36% over the comparison schemes. In such a scenario, Smart-Transfer is able to achieve significant improvement over the No-Transfer comparison scheme because long stretches during which the performance requirements are low enough to be satisfied by lower-level services occur regularly. It is also able to achieve significant improvement over the All-Transfer comparison scheme by minimizing the required number of transfers, thereby saving money by avoiding unnecessary transfers.

One limitation of the proposed Smart-Transfer scheme is its reliance on predicted re-

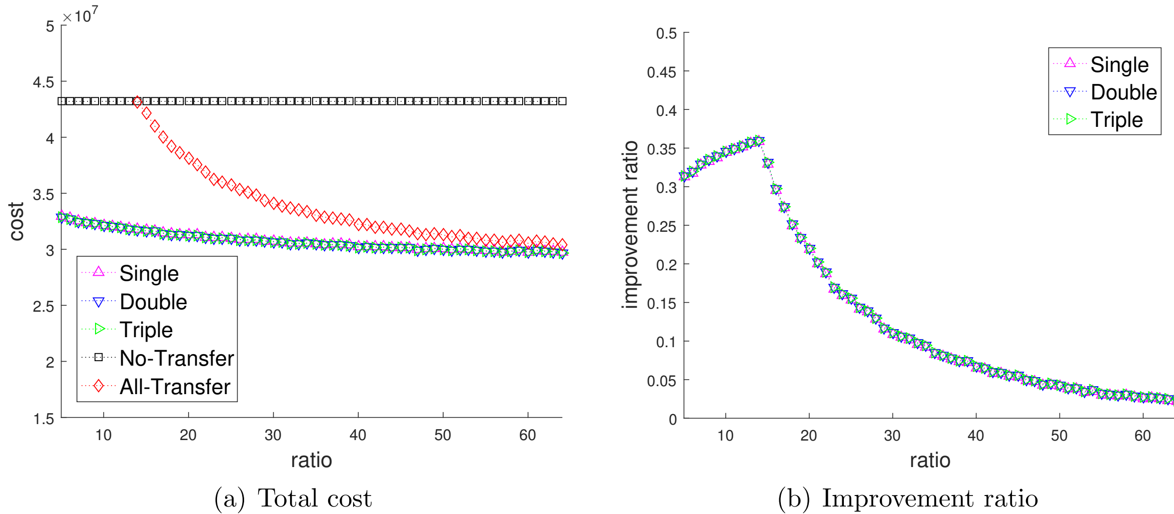


(a) 0% idle

(b) 50% idle

Figure 5.5: Improvement ratio of Smart-Transfer to the All-Transfer or No-Transfer scheme that yields a lower cost: 5 storage services are available, the ratio of data-transfer cost to data-storage cost is 12 for 5.5(a) and 28 for 5.5(b), and the total simulation time is $43,200$ time units.

quests to use as inputs. However, the scheme is able to produce good results even when the predictions used as input only look a short ways into the future. Figure 5.7 compares the cost of using an entire month's worth of predictions as input against splitting those predictions up into small chunks, solving those chunks individually, and then aggregating the costs. The figure shows that, when handling a month's worth of requests, predicting 180 minutes (3 hours) ahead is sufficient to produce a result which is only 1% more expensive than that produced by solving for the entire month.



(a) Total cost (b) Improvement ratio

Figure 5.6: Performance under a 24-hour cycle of high and low loads: 5 storage services are available, the ratio of data-transfer cost to data-storage cost is 28, and the total simulation time is $43,200$ time units.

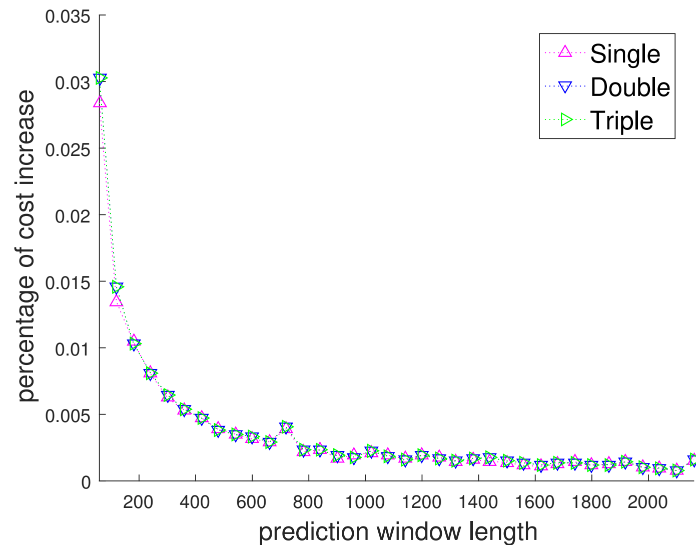


Figure 5.7: Increased cost percentage resulting from splitting a month of request prediction window into short prediction windows: 5 storage services are available, the ratio of data-transfer cost to data-storage cost is 28, and the average request duration is 15 minutes.

Chapter 6

FUTURE WORK

The Smart-transfer scheme presented in this thesis is a novel strategy for managing data in cloud storage services with the goal of minimizing costs. However, in its current form it is subject to limitations which provide opportunities to expand on Smart-Transfer by way of future work.

One significant limitation of Smart-Transfer is its reliance on predictions of future data access requests. While developing a strategy for generating such predictions is a separate research area outside the scope of this project, future work could be done regarding the relationship between Smart-Transfer and these predictions. For example, this thesis assumes that the predictions used by Smart-Transfer are accurate and does not take into account costs or performance losses associated with inaccurate predictions. Further work could be done which investigates the impact of inaccurate predictions on the performance of Smart-Transfer, as well as strategies to help make Smart-Transfer more robust with respect to such predictions.

A second avenue of potential future work would be to generalize the Smart-Transfer algorithm to be able to use any number of simultaneous storage services. Currently, Smart-Transfer is able to consider storing data at up to three storage services simultaneously. The experimental results presented in this thesis indicate that, for the scenarios which were simulated, the triple-service solution did not perform significantly better than the double-service (and sometimes the single-service) solution, indicating that adding more possible simultaneous services is unlikely to improve performance significantly. Furthermore, complexity analysis of the algorithms shows that the time complexity of the triple-service solution is significantly larger than that of the other solutions, especially when the number of avail-

able storage services is high. However, the fact remains that, because they only consider a limited number of possible simultaneous storage services, none of the three versions of Smart-Transfer presented in this thesis are guaranteed to produce an optimal solution in all scenarios. While we believe that the results presented in this thesis indicate that going beyond a double-service solution is unnecessary from a practical standpoint, it could be of theoretical value to investigate the truly optimal strategy.

To produce such a generalized solution, however, would likely require a complete restructuring of the algorithm which computes the minimal cost transfers for each request (Algorithm 1 and its variations). Currently, the single-service version of this algorithm accomplishes the majority of its computations using a trio of nested for loops: the outer loop iterates over each request in the request sequence, the middle loop iterates over each service which is able to handle the current request, and the innermost loop iterates over each service which was potentially used to handle the previous request. The double-service variation on this algorithm adds a fourth nested for loop to iterate over all potential secondary storage services, and the triple-service variation likewise adds a fifth nested for loop (for all potential tertiary services). This approach works well enough for producing a variant solution which is able to handle a known number of simultaneous storage services. However, despite the fact that the code in the added for loops follows a clear pattern, this approach would appear to fail to be able to generalize the solution to an indefinite number of storage services without a way of dynamically adding for loops following this pattern. Thus, in order to find a generalized solution, the algorithm would likely need to be restructured significantly, possibly as a recursive function which takes as one of its arguments the number of simultaneous storage services to consider.

Another manner in which this research could be extended is to test it on actual cloud storage services rather than simply testing it in simulations. Investigation could also be performed into the typical usage patterns faced by cloud-enabled applications such as on-demand video streaming. Knowing which patterns occur frequently would allow for the improvement of simulated tests of Smart-Transfer, as simulations using such patterns would

more accurately replicate real-world scenarios.

Chapter 7

CONCLUSION

In this thesis, we investigated the issue of cost-efficient cloud storage strategies. Specifically, we addressed the problem of how to transfer data between different cloud storage services in order to only pay for needed performance guarantees while still satisfying predicted performance requirements. As a solution to this problem, we proposed and evaluated Smart-Transfer, an inter-service data storage and transfer scheme designed to reduce the total data storage and transfer cost in the cloud while adhering to the performance requirements of a sequence of data-access requests. In contrast to schemes that either persist data in one storage service or move data exactly following the users request, Smart-Transfer is unique in that it intelligently determines a sequence of transfer actions which minimizes cloud storage costs based on predicted usage. Smart-Transfer chooses this transfer sequence so as to ensure that the performance requirements of all predicted data-access requests will be satisfied, and minimizes costs by avoiding unnecessary transfers as well as overpaying for unneeded performance guarantees.

In Chapters 1 and 2, we provided an introduction of the background information related to this problem including an overview of some existing cloud service offerings as well as some example scenarios in which Smart-Transfer could be beneficial. We also described research works which were related to this thesis.

In Chapter 3, we provided a formal description of the problem addressed by this thesis. This included a description of the system model and the important components of the solution such as storage services, requests, and transfer actions; a description of the preprocessing steps which are necessary to ensure that the data-access request sequence input is properly formatted to be able to be handled by Smart-Transfer; and a formal problem

statement which mathematically described the components and constraints of the problem.

In Chapter 4, we described the specific solutions which the Smart-Transfer scheme uses to solve the problem addressed in this thesis. This included an algorithmic representation of the standard, single-service Smart-Transfer strategy along with a description of the functioning of this algorithm. Two variations - the double-service and triple-service solutions - were also described, along with the modifications to the Smart-Transfer algorithm necessary to achieve the double-service solution.

In Chapter 5, we described the results of the evaluation of Smart-Transfer and its variations. All three versions of Smart-Transfer were tested in simulations alongside two different comparison schemes, and these results were compared and evaluated. We demonstrated that, in terms of overall cost, all versions of Smart-Transfer perform as well as or better than the two comparison schemes. We also demonstrated that Smart-Transfer is able to perform significantly better than the comparison schemes, performing up to 41% better under the right circumstances.

In Chapter 6, we described several avenues of possible future work which could extend the work presented in this thesis. This included development of predictive abilities which could support Smart-Transfer, generalizing Smart-Transfer to be able to consider any number of simultaneous storage services, and implementation of Smart-Transfer using existing cloud storage services.

BIBLIOGRAPHY

[1] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. Racs: A case for cloud storage diversity. In *ACM SoCC*, 2010.

[2] Bryce Allen, John Bresnahan, Lisa Childers, Ian Foster, Gopi Kandaswamy, Raj Kettimuthu, Jack Kordas, Mike Link, Stuart Martin, Karl Pickett, and Steven Tuecke. Globus online: Radical simplification of data movement via saas, 2011.

[3] Amazon AWS. Amazon DynamoDB. https://aws.amazon.com/dynamodb/.

[4] Amazon AWS. Amazon DynamoDB FAQs. https://aws.amazon.com/dynamodb/faqs/.

[5] Amazon AWS. Amazon DynamoDB pricing. https://aws.amazon.com/dynamodb/pricing/.

[6] Amazon AWS. Amazon EBS Pricing. https://aws.amazon.com/ebs/pricing/.

[7] Amazon AWS. Amazon EBS Product Details. https://aws.amazon.com/ebs/details/.

[8] Amazon AWS. Amazon EFS Pricing. https://aws.amazon.com/efs/pricing/.

[9] Amazon AWS. Amazon Elastic File System (Amazon EFS) - Details. https://aws.amazon.com/efs/details/.

[10] Amazon AWS. Amazon Elastic File System (EFS) FAQs. https://aws.amazon.com/efs/faq/.

[11] Amazon AWS. Amazon ElastiCache. https://aws.amazon.com/elasticache/.

[12] Amazon AWS. Amazon ElastiCache FAQs. https://aws.amazon.com/elasticache/faqs/.

[13] Amazon AWS. Amazon ElastiCache pricing. https://aws.amazon.com/elasticache/pricing/.

[14] Amazon AWS. Amazon ElastiCache Product Features and Details. https://aws.amazon.com/elasticache/details/.

[15] Amazon AWS. Amazon Glacier Pricing. https://aws.amazon.com/glacier/pricing/.

[16] Amazon AWS. Amazon S3 Pricing. https://aws.amazon.com/s3/pricing/.

[17] Amazon AWS. Amazon S3 Storage Classes. https://aws.amazon.com/s3/storage-classes/.

[18] Amazon AWS. Amazon Simple Storage Service (S3) FAQs. https://aws.amazon.com/s3/faqs/.

[19] Amazon Web Services. Amazon Data Pipeline. https://aws.amazon.com/datapipeline.

[20] Amazon Web Services. Cloud Storage Services. https://aws.amazon.com/products/storage/.

[21] Michael Borkowski, Stefan Schulte, and Christoph Hochreiner. Predicting cloud resource utilization. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, UCC '16, 2016.

[22] Michael J. Brim, David A. Dillow, Sarp Oral, Bradley W. Settlemyer, and Feiyi Wang. Asynchronous object storage with qos for scientific and commercial big data. In *ACM Parallel Data Storage Workshop*, 2013.

[23] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2), June 2008.

[24] Zhangyu Chang and S.-H. Gary Chan. Video management and resource allocation for a large-scale vod cloud. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 12(5s), September 2016.

[25] Stefania Colonnese, Francesca Cuomo, Tommaso Melodia, and Raffaele Guida. Cloud-assisted buffer management for http-based mobilevideo streaming. In *ACM Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, and Ubiquitous Networks*, 2013.

[26] Khaled Diab, Tarek Elgamal, Kiana Calagari, and Mohamed Hefeeda. Storage optimization for 3d streaming systems. In *ACM Multimedia Systems Conference*, 2014.

[27] Google Cloud Platform. Cloud Datastore. https://cloud.google.com/datastore/.

[28] Google Cloud Platform. Cloud Storage Options. https://cloud.google.com/products/storage.

[29] Google Cloud Platform. Could BigTable. https://cloud.google.com/bigtable/.

[30] Google Cloud Platform. Google Cloud Storage Pricing. https://cloud.google.com/storage/pricing.

[31] Google Cloud Platform. Pricing. https://cloud.google.com/bigtable/pricing.

[32] Google Cloud Platform. Pricing and Quota. https://cloud.google.com/datastore/pricing.

[33] Google Cloud Platform. Storage Classes. https://cloud.google.com/storage/docs/storage-classes.

[34] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. The rise of big data on cloud computing: Review and open research issues. *Information Systems*, 47:98 – 115, 2015.

[35] Qin Jia, Zhiming Shen, Weijia Song, Robbert van Renesse, and Hakim Weatherspoon. Supercloud: Opportunities and challenges. *SIGOPS Oper. Syst. Rev.*, 49(1), January 2015.

[36] Qin Jia, Robbert Van Renesse, and Hakim Weatherspoon. Supercloud: Economical cloud service on multiple vendors. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, 2013.

[37] Stephen Kaisler, Frank Armour, J. Alberto Espinosa, and William Money. Big data: Issues and challenges moving forward. *2013 46th Hawaii International Conference on System Sciences (HICSS 2013)*, 00, 2013.

[38] Ian Kash, Qin Jia, Zhiming Shen, Weijia Song, Robbert van Renesse, and Hakim Weatherspoon. Economics of a supercloud. In *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, CrossCloud '16, 2016.

[39] Kim Weins. Cloud Computing Trends: 2017 State of the Cloud Survey. http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2017-state-cloud-survey.

[40] Konstantinos Giannakouris, and Maria Smihily. Cloud computing - statistics on the use by enterprises. http://ec.europa.eu/eurostat/statistics-explained/index.php/Cloud_computing_-_statistics_on_the_use_by_enterprises.

[41] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. Cloudcmp: Comparing public cloud providers. In *ACM SIGCOMM Conference on Internet Measurement*, 2010.

[42] G. Liu and H. Shen. Minimum-cost cloud storage service across multiple cloud providers. In *IEEE International Conference on Distributed Computing Systems*, 2016.

[43] H. Liu. Big data drives cloud adoption in enterprise. *IEEE Internet Computing*, 2013.

[44] Kuien Liu, Haozhou Wang, and Yandong Yao. On storing and retrieving geospatial big-data in cloud. In *ACM SIGSPATIAL International Workshop on the Use of GIS in Emergency Management*, 2016.

[45] G. S. Machado, T. Bocek, and B. Stiller. Picsmu: A system to aggregate multiple heterogeneous cloud services' storage. In *IEEE Network Operations and Management Symposium*, 2014.

[46] G. Sperb Machado, T. Bocek, M. Ammann, and B. Stiller. A cloud storage overlay to aggregate heterogeneous cloud services. In *IEEE Conference on Local Computer Networks*, 2013.

[47] Y. Mansouri, A. N. Toosi, and R. Buyya. Brokering algorithms for optimizing the availability and cost of cloud storage services. In *IEEE International Conference on Cloud Computing Technology and Science*, 2013.

[48] Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[49] Microsoft Azure. Azure Redis Cache FAQ. https://docs.microsoft.com/en-us/azure/redis-cache/cache-faq.

[50] Microsoft Azure. Azure Storage. https://azure.microsoft.com/en-us/services/storage/?b=16.50.

[51] Microsoft Azure. Azure Storage Pricing. https://azure.microsoft.com/en-us/pricing/details/storage/blobs/.

[52] Microsoft Azure. Azure Storage Pricing. https://azure.microsoft.com/en-us/pricing/details/storage/tables/.

[53] Microsoft Azure. Azure Storage Pricing. https://azure.microsoft.com/en-us/pricing/details/storage/queues/.

[54] Microsoft Azure. Azure Storage Pricing. https://azure.microsoft.com/en-us/pricing/details/storage/files/.

[55] Microsoft Azure. Data Factory. https://azure.microsoft.com/en-us/services/data-factory.

[56] Microsoft Azure. DocumentDB Pricing. https://azure.microsoft.com/en-us/pricing/details/documentdb/.

[57] Microsoft Azure. Introduction to Microsoft Azure Storage. https://docs.microsoft.com/en-us/azure/storage/storage-introduction.

[58] Microsoft Azure. Performance levels in DocumentDB. https://docs.microsoft.com/en-us/azure/documentdb/documentdb-performance-levels.

[59] Microsoft Azure. Redis Cache Pricing. https://azure.microsoft.com/en-us/pricing/details/cache/.

[60] Microsoft Azure. Request Units in DocumentDB. https://docs.microsoft.com/en-us/azure/documentdb/documentdb-request-units.

[61] Microsoft Azure. SLA for DocumentDB. https://azure.microsoft.com/en-us/support/legal/sla/documentdb/v1_1/.

[62] Microsoft Azure. SLA for Storage. https://azure.microsoft.com/en-us/support/legal/sla/storage/v1_0/.

[63] Martin Placek and Rajkumar Buyya. Storage exchange: A global trading platform for storage services. In *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference*, 2006.

[64] Murray Stokely, Amaan Mehrabian, Christoph Albrecht, Francois Labelle, and Arif Merchant. Projecting disk usage based on historical trends in a cloud environment. In *Proceedings of the 3rd Workshop on Scientific Cloud Computing Date*, ScienceCloud '12, 2012.

[65] Yang Tang, Gang Hu, Xinhao Yuan, Lingmei Weng, and Junfeng Yang. Grandet: A unified, economical object store for web applications. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, 2016.

[66] Richard L. Villars and Carl W. Olofson. Big data: What it is and why you should care. *IDC*, 2011.

[67] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. Cspan: Cost-effective geo-replicated storage spanning multiple cloud services. In *ACM SIGCOMM*, 2013.