Master thesis

Long-term fault tolerant storage of critical sensor data in the intercloud

---

ING. J.R. VAN DER TIL

Supervisor TNO: ir. J.S. van der Veen
First Supervisor University: prof. dr. ir. M. Aiello
Second Supervisor University: prof. dr. ir. P. Avgeriou

FINAL
Groningen, August 29, 2013

This work is dedicated to my dear grandfather R.J. Coops.
He died on July 8, 2013 while I was finishing my Master thesis.
He took great interest in my study and encouraged me throughout the years.
He was convinced of my capabilities and it made him proud.
I will miss his great sense of humor and support.
He was my hero and inspiration.
A great man passed away.

Abstract

Wireless sensor networks consist of distributed, wirelessly enabled embedded devices capable of employing a variety of electronic sensors. Each node in a wireless sensor network is equipped with one or more sensors in addition to a microcontroller, a wireless transceiver, and an energy source. The microcontroller functions with the electronic sensors as well as the transceiver to form an efficient system for relaying small amounts of important data with minimal power consumption. All the sensors combined in the wireless sensor network are capable of generating tremendous amounts of data.

This data has to be processed as well as stored for possible future requirements. Because storing Petabytes of data is a very specialized task, not every company wants to perform this itself. For this reason we look at the capabilities cloud computing offers to store large amounts of data. However, confidentiality, integrity, availability and performance are concerns when we rely on a single cloud provider. Also the lifetime of the data is tied to the lifetime of the chosen cloud provider.

We have improved the Byzantine fault tolerant quorum protocols proposed by Bessani et al. [1] by processing the input data as a stream instead of a large block. Techniques used include encryption, erasure coding, secret sharing, and public key cryptography, to provide a way to store data in a quorum of cloud providers with a space efficiency of roughly $\frac{1}{3}$.

We provide the improved pseudocode with proofs as well as a description of the architecture and design decisions for our implementation. In our performance analysis we show that we are capable of storing up to 500 000 measurements per second on a single virtual machine. Using compression techniques and more machines will allow this number to be increased even more.

Preamble

The research in this thesis was conducted at TNO Netherlands B.V. (location Groningen) from December 2012 till July 2013. Specifically, this research was carried out in the expertise group “Service Enabling & Management” of the expertise center “Technical Sciences”.

The “Nederlandse Organisatie voor Toegepast Natuurwetenschappelijk Onderzoek”[1] or TNO for short, is a nonprofit organization in the Netherlands that focuses on applied science. It was established by law in 1932 to support companies and governments with innovative, practicable knowledge. As a statutory organization, TNO has an independent position that allows it to give objective, scientifically founded judgements. In this sense it is similar to the German Fraunhofer Society.

The mission statement of TNO is: "to connect people and knowledge to create innovations that boost the sustainable competitive strength of industry and well-being of society."

[1] Dutch Organization for Applied Scientific Research

Acknowledgements

This thesis would not have existed without the help of a lot of people. Even though I am solely responsible for this thesis.

First I would like to thank TNO for providing the possibility for me to conduct my final internship at their office in Groningen. In particular I extend my gratitude to Jan Sipke van der Veen for his supervision, input and feedback throughout my internship and beyond.

I am also indebted to Marco Aiello from the University of Groningen for accepting a supervisory role for my Master Thesis, as well as his input and feedback for my thesis. I would also like to thank Paris Avgeriou for reviewing my thesis in various stages.

I would also like to thank my parents for the opportunities they created for me during my childhood and life as a student. Also a big, big thanks for the encouragement, motivation, support, input and good food while I was working on my thesis.

Finally I would like to thank my girlfriend, Marga Jol, for her patience, motivation, encouragement, love, and support throughout my internship.

Without these people I surely wouldn't have succeeded in finishing this thesis.

Contents

Contents III

List of Figures V

List of Tables V

Glossary VII

1 Introduction 1
- 1.1 Data load analysis . . . 2
- 1.2 Thesis overview . . . 3

2 Data storage and processing 5
- 2.1 Data Storage . . . 5
  - 2.1.1 Object Storage . . . 5
  - 2.1.2 Database Storage . . . 6
  - 2.1.3 Archive Storage . . . 7
- 2.2 Processing . . . 7
- 2.3 Discussion . . . 9

3 Related Work 12
- 3.1 RAID . . . 12
- 3.2 Active Storage Systems . . . 13
  - 3.2.1 RAIN . . . 13
  - 3.2.2 HAIL . . . 13
  - 3.2.3 A Security and High-Availability Layer for Cloud Storage . . . 14
  - 3.2.4 MetaStorage . . . 14
  - 3.2.5 Octopus . . . 14
  - 3.2.6 NubiSave . . . 14
  - 3.2.7 RACS . . . 15
- 3.3 Passive Storage Systems . . . 15
  - 3.3.1 Secured Cost-effective Multi-Cloud Storage . . . 15
  - 3.3.2 ICStore . . . 15
  - 3.3.3 DepSky . . . 15
- 3.4 Discussion . . . 16

4 Analysis and Design 18
- 4.1 System Model . . . 19
- 4.2 Fundamental functions . . . 20
  - 4.2.1 Stream reading . . . 20
  - 4.2.2 Unforgeable signatures . . . 21

4.2.3 Checksum calculation 21
4.2.4 Stream Splitting & Merging 22
4.3 Streaming DepSky-A 23
4.3.1 Pseudocode 23
4.3.2 Proof of correctness 26
4.4 Streaming DepSky-CA 27
4.4.1 Important changes from Streaming DepSky-A 28
4.4.2 Pseudocode 30
4.4.3 Proof of correctness 31

5 Implementation **34**
5.1 Basic Components 34
5.1.1 Stream reading 34
5.1.2 Unforgeable Signatures 36
5.1.3 Checksum Calculation 39
5.1.4 Stream Splitting 41
5.1.5 Stream Merging 44
5.1.6 Metadata 48
5.1.7 Minor Components 50
5.2 Streaming DepSky-A 52
5.2.1 Write algorithm 52
5.2.2 Read algorithm 55
5.3 Streaming DepSky-CA 57
5.3.1 Write algorithm 62
5.3.2 Read algorithm 63
5.4 Future Work 64

6 Experiments **68**
6.1 Cloud Storage Performance Tests 69
6.1.1 Results 73
6.2 Streaming DepSky Local Performance Test 80
6.2.1 Streaming DepSky-A results 82
6.2.2 Streaming DepSky-CA results 82
6.3 Streaming DepSky Cloud Performance Test 84

7 Conclusion **90**
7.1 Costs of Implementation 90

References **95**

Appendix **96**

A1 Cloud Computing & Distributed Systems background **96**

A1.1 Cloud Computing 96
A1.1.1 Essential characteristics 96
A1.1.2 Service models 97
A1.1.3 Deployment models 98
A1.2 Distributed system models 100
A1.2.1 Interaction model 100
A1.2.2 Security model 104
A1.2.3 Failure model 107

A2 ACID Properties **110**
A2.1 Atomicity 110
A2.2 Consistency 110
A2.3 Isolation 110
A2.4 Durability 110

A3 SOLID principles **111**
A3.1 Single responsibility principle 111
A3.2 Open-closed principle 112
A3.3 Liskov substitution principle 112
A3.4 Interface segregation principle 112
A3.5 Dependency inversion principle 112

A4 Design Patterns **114**
A4.1 Abstract Factory 114
A4.2 Strategy 114
A4.3 Object Pool 115
A4.4 Singleton 116

List of Figures

1 The lambda architecture. 9
2 Architecture of file storage 19
3 Space efficiency as a function of f 27
4 Graphical illustration of metadata hash storage argument, failure case. 29
5 Graphical illustration of metadata hash storage argument, success case. 30
6 Basic components provided by the JDK 34
7 Class Diagram for the BlockReader 36
8 Metadata signature class layout 38
9 Class layout for checksum calculation and verification 39
10 Class layout for Stream splitting 41

11 Activity diagram for StreamDecomposerInputStream.read 42
13 Class layout for StreamComposer 45
14 Activity diagram for ComposedInputStream.read 46
15 Activity diagram for the generateNewBlock function of the Stream-Composer 47
16 Class layout for Metadata representation and serialization 51
17 QuorumExecutor class diagram 52
18 Class layout of the JClouds library and its abstractions 53
19 Class layout for the Streaming DepSky-A write algorithm 54
20 Class layout for the Streaming DepSky-A read algorithm 56
21 Class layout for the erasure coding and decoding functionality 59
22 Class layout for the encryption and decryption of streams 61
23 Class layout for the Secret Sharing Scheme 62
24 Class layout for the Streaming DepSky-CA write algorithm 66
25 Class layout for the Streaming DepSky-CA read algorithm 67
26 Test environment in the cloud. 68
27 Cloud test environment on premise. 68
28 Cloud test environment in the cloud. 69
29 Test application class layout 70
30 Classes that implement the executable tests 71
31 Activity diagram for the run method of AbstractPerformTest 72
32 Availability per HTTP verb. 74
33 Throughput per thread for the HTTP GET verb. 75
34 Throughput per thread for the HTTP PUT verb. 77
35 Response time for the HTTP DELETE verb. 78
36 Response time for the HTTP LIST verb. 79
37 Availability of Streaming DepSky per algorithm per verb. 81
38 Performance of Streaming DepSky-A per verb. 83
39 Perceived availability for the Streaming DepSky algorithms in the cloud. 84
40 Performance of Streaming DepSky-CA per verb. 85
41 Throughput of the GET verb with the Streaming DepSky algorithms in the cloud. 86
42 Throughput of the PUT verb with the Streaming DepSky algorithms in the cloud. 87
43 Response time of the DELETE verb with the Streaming DepSky algorithms in the cloud. 88
44 Response time of the LIST verb with the Streaming DepSky algorithms in the cloud. 89
45 Hierarchy of cloud service models 98
46 Venn diagram of the types of cloud computing 99
47 Real time ordering of events 102

48 Objects and principals 104
49 Security Model Enemy 105
50 Overview of possible failures 107
51 Model of communication between 2 processes 108
52 Structure of the Abstract Factory Pattern 115

List of Tables

1 One-dimensional measurement 2
2 Two-dimensional measurement 2
3 Approximated data load for various insertion rates. 3
4 Comparison of storage solution capabilities 17
5 Virtual Machine Specifications 71
6 Overall statistics per cloud provider 73
7 Availability statistics for the Streaming DepSky algorithms 80
8 Storage costs per provider 91
9 Costs of storing the dataset (150 TB) 91
10 Overview of timing failures 109

Glossary

API

Application **P**rogramming **I**nterface. 5, 7, 14, 51, 58, 61

DBMS

Data**b**ase **M**anagement **S**ystem. 7

DFS

Distributed **F**ile **S**ystem. 6

GPS

Global **P**ositioning **S**ystem. 102

HDFS

Hadoop **D**istributed **F**ile **S**ystem. 5, 8

IaaS

Infrastructure **a**s **a** **S**ervice. 97, 98

Internet Protocol

The Internet Protocol is the principal communications protocol in the Internet protocol suite for relaying datagrams across network boundaries. Its routing function enables internetworking, and essentially establishes the Internet.

IP

Internet **P**rotocol. 105, *Glossary:* Internet Protocol

IV

Initialization **V**ector. 61–64

JDK

Java **D**evelopment **K**it. 34, 60, 70

NIST

National **I**nstitute of **S**tandards and **T**echnology. 96

NoSQL

Meaning 'Not Only SQL', this term is used to describe non relational database systems that allow easy horizontal scaling, partitioning of data and usually use a weaker concurrency model. 2

NTP

Network **T**ime **P**rotocol. 102

PaaS

Platform **a**s **a** **S**ervice. 14, 97, 98

RAID

Redundant **A**rray of **I**ndependent **D**isks. 12–14

SaaS

Software **a**s **a** **S**ervice. 98

SAN

Storage **A**rea **N**etwork. 5, 11, 14

SQL

Structured **Q**uery **L**anguage. 7, 8

VM

Virtual **M**achine. 70

1 Introduction

Advances in wireless networking, micro-fabrication and integration, and embedded microprocessors have enabled a new generation of massive-scale sensor networks suitable for a range of commercial and military applications. In a not so distant future, cheap and tiny sensors may be deployed in roads, walls, machines, and our environment creating a 'digital skin' that can sense a variety of physical phenomena of interest. Currently, TNO is actively involved in several research projects aimed at monitoring our environment. Notable examples are the IJkdijk[2] project to help governments monitor the strength of levee's[3] for coastal flood prevention. Or the Sensor City Assen[4] project that helps monitor vehicular traffic to create an intelligent transportation grid.

These projects rely on the storage and processing of the data gathered from the deployed sensor networks. Clearly, the time between the gathering of the data and the processing should be as low as possible for these systems to maximize their potential. However, we also want to minimize the costs of storing and processing the data. Clearly this is a challenge.

Wireless sensor networks consist of distributed, wirelessly enabled embedded devices capable of employing a variety of electronic sensors. Each node in a wireless sensor network is equipped with one or more sensors in addition to a microcontroller, wireless transceiver, and energy source. The microcontroller functions with the electronic sensors as well as the transceiver to form an efficient system for relaying small amounts of important data with minimal power consumption.

Unlike current information services such as those on the Internet where information can easily get stale or be useless because it is too generic, sensor networks promise to couple end users directly to sensor measurements and provide information that is precisely localized in time and/or space, according to the user's needs or demands [2]. If these needs or demands can be fitted in a computational model, then it is possible to provide a live view to the users by feeding data directly from the sensor network into the model.

From a data storage point of view, it is possible to view the sensor network as a distributed database. However, to keep the price of sensor nodes low they are usually not fitted with large volatile or permanent storage, making long term storage of measurements a challenge. This calls for another way to store the data collected by these sensors. Of course, it is possible to simply insert all the data generated by the sensor network into a database. But would we use a traditional relational database?

[2] `http://www.ijkdijk.nl`
[3] Or dike, floodbank, or in dutch: "Dijk"
[4] `http://www.sensorcity.nl`

Or maybe one of the newer NoSQL alternatives? Maybe we could utilize cloud computing to relieve us from the administrative and technical burden of maintaining our own database systems. But how do we deal with the problems that are unique to cloud computing?

Since cloud computing offers virtually endless and scalable storage that requires little to no upfront financing, this might be an ideal solution to store large amounts of sensor data. Then we can shift the administrative and technical burden of maintaining the storage to the cloud provider, while only paying for the actual storage consumed. There are however some disadvantages when storing data in the cloud, such as limited availability, data lock-in, reduced data confidentiality & auditability, and data transfer bottlenecks [3]. Another downside is that the lifetime of the stored data is linked to the chosen cloud provider.

TNO also recognized these concerns but believes that a solution might lie within the use of the Intercloud, a federation of multiple cloud providers. Therefore, in this thesis we will investigate how we can manage measurements from sensor networks to enable long term storage of these measurement, while also allowing the data to be used to provide computational models with a continuous stream of measurements from the sensor network. As requested by TNO, we will research the use of storage or compute facilities offered by multiple cloud providers to overcome the disadvantages of using a single cloud provider.

1.1 Data load analysis

In order to understand the amount of data generated by a single sensor network we need to make some assumptions about the size and frequency of incoming measurements. Since we don't want to enforce a fixed data model on the data source, our storage facility should be able to handle very diverse formats of the measurements stored. That said, however, we do assume that all sensor measurements contain two fields: a *SensorId* that uniquely identifies the sensor and a *TimeStamp* which indicates when the measurement was taken. Since most sensors will only report a one- or two-dimensional value we can assume that most sensor measurements match the data model in either Table 1 or Table 2.

Field	**Data type**
SensorId	GUID
Timestamp	long
Value	double

Table 1: One-dimensional measurement

Field	**Data type**
SensorId	GUID
Timestamp	long
ValueX	double
ValueY	double

Table 2: Two-dimensional measurement

To understand the amount of storage our system needs, we first need to analyze the amount of data that is generated and at what pace this data is generated.

The frequency at which a sensor reports a measurement can vary a lot. For example, a vibration sensor that measures the vibrations generated by a certain amount of traffic over a bridge might range in the tens of thousands of measurements per second. While a sensor that measures the temperature or atmospheric pressure might only report a measurement every 15 minutes or even less often.

We assume that the sensor network will consist of 10 000 sensors, and that each sensor generates measurements in at most 2 dimensions. Each measurement contains the fields as described in Table 2. So a single measurement will be at most:

$$S_m = 16 \text{ (SensorId) } + 8 \text{ (timestamp) } + 8 \text{ (value x) } + 8 \text{ (value y) } = 40 \text{ bytes}$$

In Table 3 we have outlined the expected incoming data load for various measurement intervals.

	1 / min	**4 / min**	**1 / sec**	**10 / sec**	**100 / sec**
MiB[5] / second	0,006	0,025	0,40	4,0	40
GiB[6] / year	200	800	12 000	120 000	1 200 000

Table 3: Approximated data load for various insertion rates assuming 40 bytes per measurement and 10 000 sensor nodes.

According to our calculations, we can expect incoming network bandwidth to range between 6 KiB/sec up to 40 MiB/sec, and our expectation is that in the future this will only increase. We observe a lot of repetition in the data, such as the *SensorId* and *Timestamp*, compression techniques could allow us to significantly reduce the size of the data, at the cost of some computational effort. This will be beneficial when the datasets grow into the tera- or petabyte scale, even a reduction of 10-20% can yield a significant cost benefit. It is thus advisable to apply compression techniques to the data before it is stored. Obviously, reducing the amount of data stored in other ways is also advisable.

1.2 Thesis overview

In Chapter 2.1 we will cover the various storage and processing methods of data in use today, from these we will select a storage method which we will research. We discuss work related to this thesis in Chapter 3. We propose an improved storage algorithm in Chapter 4 and discuss our implementation in Chapter 5. We will cover

[5] 1 MiB is 2^{20} bytes.
[6] 1 GiB is 2^{30} bytes.

the performance and reliability tests in Chapter 6. Finally we discuss our results in Chapter 7 while also providing a cost estimation of storing a real world data set.

For readers that are not familiar with the cloud computing and distributed systems domains, we included additional background information in appendix A1.

2 Data storage and processing

2.1 Data Storage

Electronic data storage needs continue to grow as companies produce more information in electronic formats every day, making storage space increasingly important. Managing data storage for performance, integrity, and scalability is one of the big challenges in Information Technology management and planning. For our purposes we categorized data storage as: object storage, database storage and archive storage. Each of these storage methods has its own requirements in terms of availability, scalability, performance, as well as price.

2.1.1 Object Storage

By object storage we mean the storing of files in a storage system so that files are accessed through an **A**pplication **P**rogramming **I**nterface (API). This allows us to operate on these objects without knowledge of the underlying storage method. Files are referenced by a unique identifier, instead of a location on a disk [4]. An example of object storage that is widely used in companies, is a logical disk that is stored in a **S**torage **A**rea **N**etwork (SAN). This logical disk is connected to a host using a fiber channel connection such as InfiniBand, or the IP based iSCSI. The disk is identified using a unique Logical Unit Number (LUN), and is accessed through the SCSI protocol as if a local file system. Commonly, we are not really interested in searching inside the objects themselves. We usually can determine which object we need by looking at its filename or identifier, or if available, any metadata that is stored alongside the object.

Another example of a object storage system is implemented by the Apache Hadoop framework. This framework is based on the Map-Reduce paper [5] published by Google. In this paper the authors describe a method to simplify the processing of large data sets (in the order of multiple PetaBytes[7]) on large computer clusters. As an example they present two programs, one that searches through a large file set for a specific pattern, and another that sorts a TeraByte of data. The **H**adoop **D**istributed **F**ile **S**ystem (HDFS) [6] is the file system component of Hadoop, and is essentially an object store. Files that are stored in HDFS are distributed across multiple nodes for durability, and can even be split into multiple parts transparently. DataNodes hold the blocks of data that are assigned to them, while the NameNode holds the metadata for all files. Hadoop applications can then use the HDFS API to find and access relevant files, and also use the HDFS to store their results.

[7] 1 PetaByte is 1000 TeraBytes

Note that while both examples given are implemented as a **D**istributed **F**ile **S**ystem (DFS), and indeed, most object storage methods are a DFS. The standard file system used by computer users around the world conceptually maps very close to an object storage system. Files are also a combination of data (the contents) and metadata (for example, the filename), the same applies to objects. However, a normal file system (non distributed) uses the location on the disk to locate files. This does not apply to object storage.

2.1.2 Database Storage

Database storage operates on data stored in files that have been structured according to a certain schema or format. This is different from object storage as the object storage system does not care about the structure of the files. Database storage provides rich and powerful ad hoc querying functionality. Usually, it also provides strong consistency and integrity properties. These properties are known as the ACID properties[8]. Of the database systems in use today, the relational database is the oldest and most commonly used database system. Edgar Codd can be considered the father of the modern relational database systems. In [7] he proposed a new system for storing and working with large databases. But unlike the in that time commonly used navigational database, which used a sort of linked list of free-form records, he proposed to use a table of fixed-length records, with each table used for a different type of entity.

The main disadvantages of the linked-list system was that it was very inefficient at storing "sparse" database, in which parts of records could be left empty. Codd solved this problem by moving optional data into a separate table, where it would only take up space if required. Data in different tables would be linked together using a couple of different relationships. These relationships are one-to-one, one-to-many, many-to-one and many-to-many. The keys on which these rows would be linked are called Foreign Keys. The process called "normalization", guides a database designer in separating various entities into different tables.

The relational database was a huge success, and companies started moving all their data into Relational Database Management Systems. However, this caused a couple of problems, because not all data is suitable to be stored in a relational database system. Not all queries can be run efficiently on relational data, and due to the design of these database systems they are not the best choice in all cases. Even today, developers when confronted with a data storage problem will, by default, tend to use a relational database system. Luckily, in the last couple of years a host of new database systems are gaining popularity in the developer community. Besides rela-

[8] For more information on the ACID properties, please refer to appendix A2.

tional databases we can also distinguish, key-value stores, document stores, column oriented databases and graph databases. The considerations that should be taken before choosing one or the other still apply, and choosing one has only become more difficult.

2.1.3 Archive Storage

Archive storage has very different requirements than either object storage or database storage. Since access to the archive should be sporadic, it is not required that all the data is accessible in milliseconds. Thus, traditionally, archive storage is done in tertiary or off-line storage media, such as a tape library or optical disks. The choice for storing on tape or optical disks is because they have a longer expected lifetime than hard disks. Although optical disks degrade over time, data remains retrievable in most cases. Hard disks on the other hand tend to loose the magnetic charge on the disks over the course of time. The disadvantage is that access times are high, usually a couple of minutes to a couple of hours. In the case that optical disks are not stored on site, it can take a couple of days to access the data. The advantage is that the storage costs are very low, and that storage space should be abundant.

An example would be a large sensor data set of several PetaBytes that once processed results in a smaller set of a couple of TeraBytes. The smaller set is much more frequently accessed and thus has very different availability and scalability constraints. The large data set is not really of interest, and not often accessed. A year later the research is conducted again and with new methods a comparison is to be made between the old data and the new data. The large data set was not accessed for a year, but should be retrievable for processing once again.

2.2 Processing

Data is stored with a purpose, which is usually the act of allowing the data to be processed to gain deeper insights. With database storage the processing of the data is tightly integrated inside the **D**ata**b**ase **M**anagement **S**ystem (DBMS). For relational database systems the industry standard for querying the data is using **S**tructured **Q**uery **L**anguage (SQL). Other database systems, such as document stores and column oriented databases, provide their own querying API. As databases grow in size, various NoSQL database systems also provide the MapReduce paradigm for querying large databases efficiently.

Object storage however, does not provide querying capabilities out of the box. Besides simple file listings, users should rely on other tools to process data stored in

these systems. A commonly used tool is the Hadoop framework[9]. As discussed earlier, Hadoop comes with a object storage method out of the box. However, the use of HDFS might not be suitable for everyone. For this reason it is possible to write adapters, or plugins, to connect Hadoop to various other file storage methods.

There is however a big difference between the use of SQL, or another database storage querying method, and the use of Hadoop. With SQL it is possible to calculate results including all the data that is available up to the point at which the query was issued. Typical SQL queries are completed within a second. With a significantly large data set and Hadoop, the run time for a query can be several hours or days. Obviously, this is not suitable for all use cases. A common situation is that Hadoop is used to calculate a view of the data set, which can be queried significantly faster. However, this form of *batch processing* does not provide constant up to date information. For example, if a batch is scheduled to be run once a week, data submitted to the storage within the time between batches is not visible in the view.

To mitigate this issue we need to combine two different methods of processing data. The first is batch processing as discussed earlier, which should process the total data set at regular intervals. The second is stream (or real time) processing, such as Storm[10], which will compensate for the data that has not yet been batch processed. Both the batch and stream processing will publish views which can be queried and combined by a client. This architecture is presented in [8] as the "Lambda Architecture". A graphical view of this architecture is shown in Figure 1 (page 9). Important is that once data has been batch processed, it can be removed from the stream processed view. This allows the stream view to contain data for several hours instead of years. Accuracy is also a key factor, very accurate algorithms usually take more time in processing than approximated algorithms. A consideration could be to use slow but accurate algorithms in the batch processing, and faster approximate algorithms during stream processing.

In the Lambda architecture an important part is played by the "All data" storage. This is the single point of truth in the application. This is forced by using an append only database or data store. Once data is written, it can not be erased. A compensation action should be taken to remove this data. This allows for easy recovery when faulty data is written to the data store.

To illustrate this consider a simple example of counting how many friends one has on Facebook. There are two ways to do this, we could store an integer which is incremented each time a friend is added and decremented whenever a friend is removed. Now consider that due to a bug in the software the counter is not decremented when a friend is removed. Now consider that we can also store events such

[9] `http://hadoop.apache.org/`
[10] `http://storm-project.net/`

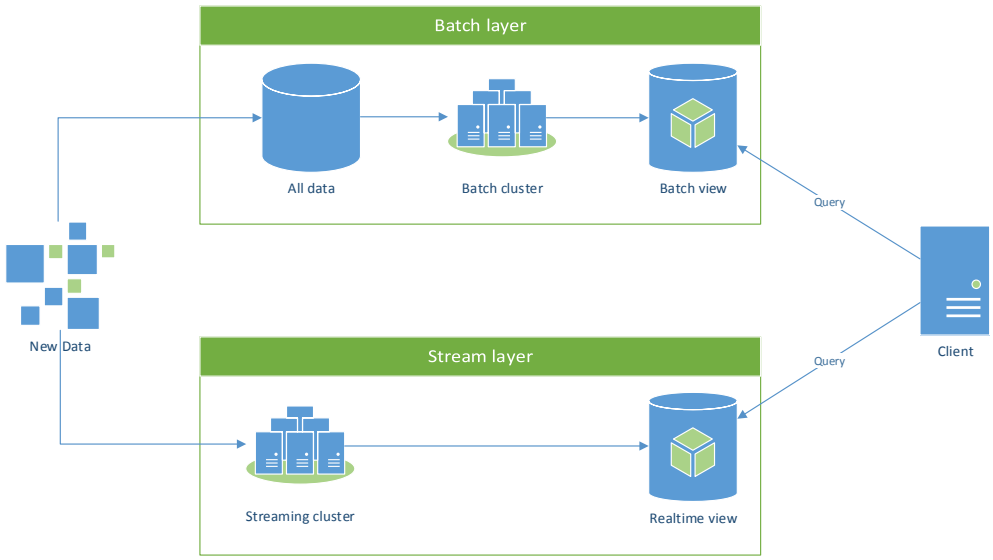


Figure 1: The lambda architecture. Image adapted from [8].

as 'Became friends with X' and 'Is no longer friends with Y'. Then we can keep a counter that is incremented each time we see the 'Became friends with' event and decrement it when we see 'Is no longer friends with'. This way we can always re-calculate the number of friends, which is not possible with simply incrementing and decrementing a database counter. It does require that the data is never thrown away or lost.

2.3 Discussion

In this chapter we introduced three types of storage that we consider the most likely candidates for our storage solution. Clearly, we require a form of archive storage to meet our long term storage requirements. We do not want to maintain the storage ourselves however, thus we want the data to be stored by a third party. Since data storage is a fundamental component of cloud computing, we can use the various public cloud offerings to store the data. Unfortunately, we do not trust a single cloud provider to safely store our data. Thus we require sufficient alternatives for a certain storage method.

Currently, within cloud computing the only real archive storage implementation is offered by Amazon Glacier. Glacier offers a very low cost storage system, but query and retrieval times are high (usually several hours). Data stored in Glacier should not be frequently accessed, or modified as it is not designed to be used this way. For frequently changing data users should use the Amazon Simple Storage Service (S3), an object storage method. Since only Amazon offers a archive storage method,

we can not use this method of storage in our solution. We also note that OVH is currently beta testing a low cost cloud archive storage method[11], however we do not consider it at this time due to its beta status. This leaves us the choice between object storage and database storage.

Database storage is offered in various forms, from simple pay-per-use storage to fully managed database clusters. For relational databases Windows Azure SQL Server is the only pay-per-use offer. All other offerings, such Amazon Relational Database Service (RDS), Amazon RedShift and Google Cloud SQL are billed per hour. These are actually managed database clusters with a convenient and easy to use API. The prices per hour vary depending on the size of the instance (servers) deployed behind the scenes. Almost all relational database system offerings have an upper limit on the size of the database, the only notable exception is Amazon RedShift which is designed as a data warehousing solution. For Amazon RDS the limit is 3 TeraBytes, for Windows Azure SQL Server 150 GigaBytes, and for Google Cloud SQL 100 GigaBytes.

NoSQL databases are also offered as pay-per-use storage and fully managed clusters. Amazon's DynamoDB is a key-value storage which has no upper limit on storage. It can be provisioned to store as much as is needed for the application. Microsoft's alternative is Windows Azure Tables, a slightly different implementation of a document store. Azure Tables can scale up to 200 TeraBytes[12] of storage for a single storage account, of which the user has a initial soft limit of 5. By contacting support this limit can be increased, however the storage limit per account can not be increased. There are also smaller companies that offer managed environments of RavenDB and MongoDB, two popular document database offerings. These are usually hosted inside a single cloud provider, most notably Amazon EC2. Scalability is thus not really limited, as starting more machines in a sharded architecture can increase the storage as needed.

Obviously, these virtualized environments impact the performance of the databases run inside them as discussed in [9]. Both [9] as well as [10] make a strong case for running Cassandra, a column oriented NoSQL database, instead of document databases as RavenDB or MongoDB. Indeed, [10] deals with a very similar problem to the one we face in this thesis. However, we differentiate because we are interested if it possible to decrease storage costs by not using active servers in the cloud to store the data. Ideally, when we are dealing with a data set that does not require 24/7 analytical capabilities we can remove actively running machines when they are not needed. This should reduce the cost of storing the data significantly. Also

[11] `http://www.ovh.nl/cloud/archive/`

[12] Slightly hard to find as the official documentation seems to be outdated. This blog post by the Azure Storage team gives the new scalability targets for storage accounts: `http://blogs.msdn.com/b/windowsazure/archive/2012/11/02/windows-azure-s-flat-network-storage-and-2012-scalability-targets.aspx`.

reducing power consumption aligns with the growing trend of Green Information Systems, which aims to reduce the amount of power current information systems and networks consume[11].

Finally, object storage, a major component of cloud computing, as for many people the main purpose of cloud computing is, indeed, storage [12]. Thus, many offerings of various providers exist. There are traditional SAN based solutions such as offered by GoGrid[13]. As well as solutions that use a web service as the primary interface, such as Amazon Simple Storage Service (S3)[14], and Windows Azure BLOB Storage[15]. The major benefit is that cloud object storage is billed by GigaByte of storage (or network bandwidth) consumed. Pricing can be tiered, as done by Amazon and Windows Azure, or a flat price per GigaByte regardless of the amount stored. Capacity scaling is usually not limited, or the limits are very high[16]. Another major advantage is that no servers have to be maintained by the user, all storage and maintenance is done by the cloud provider. The obvious disadvantage is that no querying or analytical capabilities are offered out of the box, but as we discussed earlier this can be resolved.

We will thus investigate how we can use cloud object storage to provide a cost efficient storage solution, which is sufficiently protected against single cloud provider outages, and reduces vendor lock-in and data lock-in, while improving the confidentiality of the data stored in the cloud.

[13] `http://www.gogrid.com/products/infrastructure-cloud-storage`
[14] `http://aws.amazon.com/s3/`
[15] `http://www.windowsazure.com/en-us/services/data-management/`
[16] Windows Azure Storage is a notable example, as the storage for a single account is capped at 200 TeraBytes

3 Related Work

We are looking for a system that can store data for prolonged amounts of time and is highly available. Traditionally servers would be equipped with a number of disks and expose these through the network to other systems. However, this has certain downsides. When a disk would fail, the data stored on it would be lost and thus, backups would be very important. As the number of disks exposed grows, the chance that one of these will fail increases. As an example: Yahoo! reports that in their Hadoop cluster of 3500 machines they experience a failure in 0.8% of the nodes each month [6]. That means that the data on roughly one node per day is lost.

3.1 RAID

To deal with the loss of individual disks in a single system, the disks are usually configured as a **R**edundant **A**rray of **I**ndependent **D**isks (RAID) or simply, RAID array [13]. Each RAID array is configured to a certain RAID level. The RAID level defines which failures the array can tolerate, and which performance characteristics the user can expect. For example, a RAID level of 0 (written as RAID-0) means that data is striped over multiple disks. This means that when a block of data is written to a RAID-0 array, that the block will be split in n parts (where n is the number of disks in the array) and each disk will store a single part. Effectively increasing the read and write performance by a factor n, while the space efficiency[17] is 1. The downside of RAID-0 is that if a single disk in the array fails, all the data stored in the array is lost. Clearly not ideal for all storage needs, which is why there are lots of different RAID levels.

Most commonly used in enterprise environments are RAID-1 and RAID-5. Although, RAID-5 is being superseded by RAID-6 to deal with the increasing size of hard disks. RAID-1 uses mirroring without parity or striping, and can tolerate $n-1$ disk failures. Read performance is amplified by a factor n, write performance is not affected, and the space efficiency is $\frac{1}{n}$. RAID-5 uses striping with distributed parity that can tolerate the failure of a single disk in the array. The read and write performance is increased by a factor $n-1$, while the space efficiency is $1-\frac{1}{n}$. RAID-6 uses striping with double distributed parity, so that it can tolerate the loss of two disks in a single array. RAID levels can also be nested to combine advantages and disadvantages of the various raid levels[18].

[17] This is a factor, thus 1 is optimal.

[18] For more information on RAID and RAID levels, we refer the reader to the excellent article on Wikipedia: `http://en.wikipedia.org/wiki/RAID`.

The RAID techniques used in storage systems are also used when applied to distributed storage systems. The various papers we will discuss promote the usage of striping, mirroring (or replication) and erasure coding (or parity) to achieve various performance and reliability constraints. We distinguish two different categories within the papers. The first set of papers propose systems that can perform calculations on the storage end, or require the use of systems other than the storage systems to perform their storage function. The second set of papers propose systems that perform any required calculations on the client and simply use the storage to store data without requiring additional calculations. We will refer to first set of papers as: "Active Storage Systems", and to the second set of papers as: "Passive Storage Systems".

3.2 Active Storage Systems

3.2.1 RAIN

The authors of [14] propose a Redundant Array of Independent Net-storages (RAIN). This system splits a file into an arbitrary amount of segments in such a way that each segment does not compromise confidentiality. By keeping the distribution of segments and the relationships between distributed segments private, the original data cannot be reassembled. Data can then be stored using one or several cloud storage providers. They propose to organize the elements in their distributed cloud architecture as a traditional multi-tier Command and Control botnet. By introducing a new type of cloud service provider they move all processing into the cloud, but this does mean that the provider has to be trustworthy.

3.2.2 HAIL

In [15] the authors introduce a distributed cryptographic system called High-Availability and Integrity Layer (HAIL). HAIL allows a set of servers to prove to a client that a file is stored and retrievable. To do this it uses Proofs of Data Possession and Proofs of Retrievability. These are challenge-response protocols that allow a server (or cloud provider) to prove to a verifier (the client) that files are stored intact and are retrievable. Files stored in HAIL are protected against an active (e.g. can corrupt servers and file blocks), and mobile (e.g. can corrupt any server over time) adversary. It is however required that clients verify the contents of the server at regular intervals.

3.2.3 A Security and High-Availability Layer for Cloud Storage

A practical solution to help deal with the complexities of choosing a cloud storage provider, is proposed in [16]. When choosing a cloud storage provider stakeholders are confronted with various risks, such as data security, service availability, data lock-in, lack of Quality of Service standardization, and various legislative issues. The authors propose a system that helps deal with these issues by using RAID-like techniques to distribute data over multiple cloud providers. By abstracting cloud provider specific APIs and providing one unified API they want to avoid data lock-in. The system is exposed to an end user through a web based interface, as well as a API for interaction with other systems.

3.2.4 MetaStorage

In [17] a federated cloud storage system called MetaStorage is introduced. It is designed as a highly available and scalable distributed hash table that replicates data on top of diverse storage services. It abstracts away a lot of different storage methods, including cloud storage, SSH file servers and local filesystems by unifying them in a single API.

3.2.5 Octopus

The authors of [18] propose an new cloud service, called Octopus, that runs within **P**latform **a**s **a** **S**ervice (PaaS) providers such as Google App Engine. It allows users to import their credentials for Amazon S3 compatible services, which are then managed by the Octopus service. Currently it only supports RAID-1 like behavior, by mirroring data across the storage accounts supplied by the user. There are plans to implement RAID-5 like behavior, but it is unclear when this will be completed.

3.2.6 NubiSave

In [19] the NubiSave prototype is introduced that can match various user supplied criteria to store data in a set of cloud storage providers. The cloud storage providers are selected based on the criteria given by the user. These criteria are categorized in three groups: Quality of Service (QoS), Business, and Technical & Domain-specific. A couple of examples: Availability (QoS), Throughput (QoS), Price per storage unit (Business), capacity (Technical), redundancy (technical), etc. It is implemented as a Linux Filesystem in Userspace (FUSE) interface, so this solution maps closely to a SAN.

3.2.7 RACS

In [20] the authors introduce a Redundant Array of Cloud Storage (RACS), which is a cloud storage proxy that transparently stripes data across multiple cloud storage providers. It reduces the one-time cost of switching storage providers in exchange for additional operational overhead. The proxy is run on a separate server (or a group of servers coordinated by ZooKeeper[19]). It is implemented in Python and the source code is available online[20].

3.3 Passive Storage Systems

3.3.1 Secured Cost-effective Multi-Cloud Storage

In [21] a formal model of a system describing a cost effective storage solution using multiple cloud storage providers. It takes into account various constraints on budget and availability to distribute data optimally across a selection of storage providers. By dividing and distributing customers data, the model shows its ability of providing a customer with a secured storage under his affordable budget. It sounds as a model that NubiSave [19] could have implemented, but the authors of [19] make no reference to [21].

3.3.2 ICStore

The ICStore library proposed in [22], offers a key-value store to the client with simple read and write operations. The back end stores the files in multiple cloud storage services transparently. The library consists of multiple layers that provide guarantees about the Confidentiality, Integrity, Reliability and Consistency of files stored with it. As [22] is a research report, the implementation presented is very premature.

3.3.3 DepSky

The authors of [1] introduce two algorithms, DepSky-A and DepSky-CA, that allow users to create Byzantine fault tolerant cloud storage, by using a quorum of cloud storage providers. The algorithms are supported by theoretical proof, as well as an evaluation of the performance. The authors restrict themselves by utilizing existing cloud storage providers, and thus do not require code to be executing on the storage

[19] `http://zookeeper.apache.org/`
[20] `http://www.cs.cornell.edu/projects/racs/`

servers. They present their solution as a library that can be integrated into existing or new applications.

3.4 Discussion

All the discussed solutions provide fault tolerance, availability, confidentiality and take care to prevent vendor or data lock-in. Our ideal solution requires no running servers in the cloud, and are capable of using cloud object storage as it is implemented today. This means that solutions that require active servers for storing the data, or that require servers that act as intermediaries between the client and the actual storage, are at a disadvantage. Clearly, solutions that do not support 'traditional' cloud object storage are also not usable. A comparison table is given in Table 4.

RAIN and HAIL are designed as a storage solution in which data is stored, and they require quite a significant amount of running servers to operate. Since calculation on the servers storing the data is required for both solutions, these are unfortunately not usable.

All the other solutions are thus acceptable candidates. However, we find that one solution in particular stands out, namely DepSky ([1]). Instead of providing an entire system for storing data, it simply provides a couple of algorithms that can be used to store data in a Byzantine fault tolerant way. This allows us to create an architecture around these algorithms, that is optimized for our use case. All the other solutions provide a complete storage system, that might or might not fit our purposes. If we choose one of them, we are quite constrained in the amount of adjustments we can make. The downside of choosing DepSky is that we have to take great care when implementing the algorithms, so that the provided theoretical proofs are still valid.

Solution	Requires active storage	Requires active intermediaries	Supports current cloud storage
RAIN	X	X	
HAIL	X		
High-Availability Layer		X	
MetaStorage		X	X
Octopus		X	X
NubiSave		X	X
RACS		X	X
Multi-Cloud Storage			X
ICStore			X
DepSky			X

Table 4: Comparison of storage solution capabilities

4 Analysis and Design

We feel that the DepSky algorithms proposed in [1] can not deal with files of any significant size on a normal machine. This is due to the fact that they process the entire file in a single pass. Due to this, files have to be read into memory completely before processing can start. Also corruption of the file can only be detected by fully downloading and (attempting to) restore the file. Considering that most cloud providers bill bandwidth per GigaByte consumed, this is not optimal for large files.

For example, we can also consider optical images as a form of sensor input, such as video cameras to monitor certain geographical locations. Video files are often much larger than other data files. If we want to store this data using the original DepSky algorithms, we would require significant amounts of addressable memory in order to process these files.

Therefore, we propose improved algorithms based on those given in [1]. The major change is that instead of dealing with an single block of data we consider a sequence of data elements that are made available over time, a stream. These data elements will typically be bytes[21]. However, for efficiency we typically do not want to operate on individual bytes. Therefore, blocks of data are read from a stream and can be written to another stream, resulting in a stream of blocks of bytes. The notation we will use for a stream is:

$$x = [x^0, x^1, x^2, \ldots, x^n]$$

Which shows a stream x with data elements x^0 till x^n, these data elements are typically bytes or blocks of bytes. Note that while we give three starting elements (x^0, x^1, x^2), the restriction of n is $n > 0$.

The metadata stored alongside each file has also been changed. Before blocks are written to the output, a checksum is calculated by a cryptographical hash function and stored in the metadata. By calculating the checksum just before writing the processed block, we can directly verify the integrity of each block read before processing. This avoids wasting processing on blocks that turn out to be corrupted. The approach of calculating a checksum per block is of course not new. It is also used in the metadata files used by the BitTorrent protocol [23].

The algorithms we propose are aimed at fulfilling the role of the "All Data" component in Figure 1 (page 9). Since this should be an append only data store, we do not consider versioning of files or concurrent writing to a single file. However, should a versioning or locking scheme be required in the future, there is no reason why the strategies proposed in [1] will not work for our algorithms.

[21] Integers $i \in \mathbb{N}$ ranging $0 \leq i \leq 255$.

4.1 System Model

The system model for our algorithms is the same as the original described in [1]. For clarity, we will reproduce it here.

We assume an asynchronous distributed system composed of three parties: readers, writers and cloud storage providers. In Figure 2 (page 19) the writer is the "Sensor Server", and the storage clouds are the four clouds. The readers and writers are roles of clients, not necessarily different processes.

Readers can fail arbitrarily, i.e., they can crash, fail intermittently and present any behavior. Writers are only assumed to fail by crashing. We do not consider that writers can fail arbitrarily because, even if the protocol tolerated inconsistent writes in the replicas, faulty writers would still be able to write wrong values in data units, effectively corrupting the state of the application that uses the DepSky algorithms. Moreover, the protocols that tolerate malicious writers are much more complex with active servers verifying the consistency of writer messages, which cannot be implemented on general storage clouds.

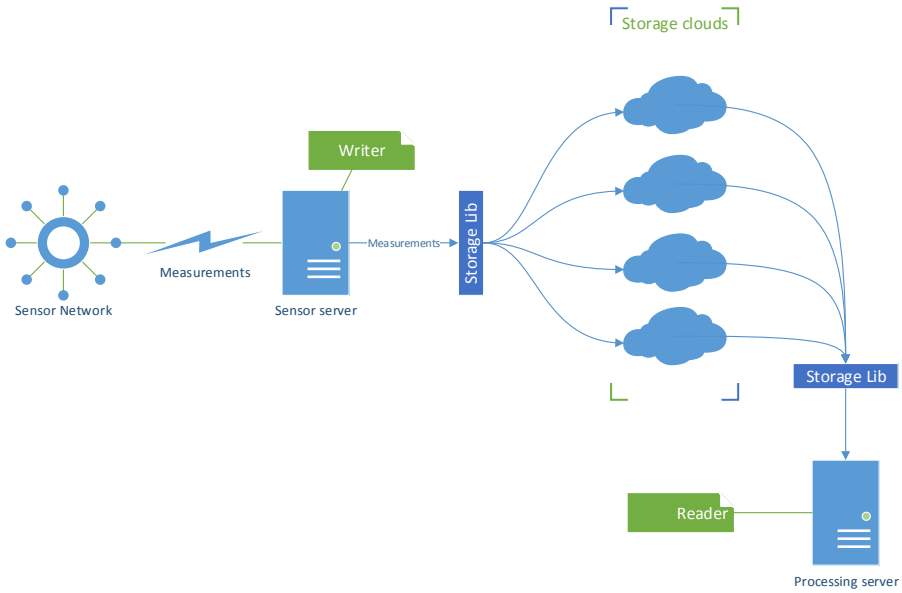


Figure 2: Architecture of file storage. Includes data source for clarity.

Each cloud is modeled as a passive storage entity that supports four operations: *list*, *get*, *put*, *delete*. A passive storage entity means that no protocol code other than what is needed to support the mentioned operations is executed. We do not allow explicit creation of containers as the original model does, however it is possible to emulate a directory structure with the filename. We assume that access control is provided by the system in order to ensure that readers are only allowed to invoke the list and get operations.

We also assume that the communication channels between the cloud and the readers and writers are reliable and secure. Thus that any message that is sent is eventually delivered to the recipient, and that the message received is identical to the one sent, and no messages are delivered twice. Also each reader and writer knows reliably the identity of the clouds, thus they are able to verify that they are communicating with the correct server. The secure channel also ensures the privacy and integrity of the data transmitted across it.

Clouds are not trusted individually, and are assumed to fail in a byzantine way. Thus, data stored can be deleted, corrupted, created or leaked to unauthorized parties. The algorithms require at least $n \geq 3f + 1$ cloud providers [24] where f is the number of faulty clouds.

There is however a difference in our model and that of [1]. Because the original algorithms have all the data in memory, it is very easy to invoke the Remote Procedure Calls (RPCs) of the cloud providers multiple times until success or until cancelled. We, however, do not have all the data in memory and can thus only invoke the write RPC once until success, failure, or cancellation. Upon failure, we can not recover. This only holds true for the writing of the file, metadata is all held in memory and we can thus invoke the write RPC for metadata multiple times.

Some cloud providers, such as Windows Azure, offer a method of uploading a file split into blocks. The cloud provider would hold on to the blocks for a certain time, and commit them when asked by the client [25]. This could allow us to circumvent this limitation, but as not all cloud providers support this we will not investigate this at this time.

For more information on distributed systems theory, please refer to Appendix A1.

4.2 Fundamental functions

The functions described in this section form the basis for the Streaming DepSky-A and Streaming DepSky-CA algorithms.

4.2.1 Stream reading

To optimize processing efficiency, we want to process blocks of bytes instead of one byte at a time. Therefore, we require a function r that transforms a stream of bytes x into a stream of blocks x'. Blocks β are of size λ. If the size of the stream is not a multiple of λ then the size of the last block δ is:

$$\mu = |\delta| = |x| \bmod \lambda$$

Thus, the function r:

$$x' = r(x, \lambda) = [\beta_0, \beta_1, \beta_2, \ldots, \delta]$$

4.2.2 Unforgeable signatures

To verify the integrity of data we will use unforgeable signatures. All writers of a set of files share a common private key K_{pr} which is used to calculate a signature σ of some data:

$$\sigma = sign(data, K_{pr})$$

Readers have access to the corresponding public key K_{pu} which is used to verify some data:

$$verify(data, \sigma, K_{pu}) = \begin{cases} \top & \text{if the signature is valid} \\ \bot & \text{if the signature is invalid} \end{cases}$$

This is not different from [1].

4.2.3 Checksum calculation

To ensure the integrity of the data we require a cryptographic hash function H, which can be used to calculate a checksum c from a block β:

$$c = H(\beta)$$

And to verify the integrity of a block:

$$H(\beta, c) = \begin{cases} \top & \text{if the block is valid} \\ \bot & \text{if the block is corrupt} \end{cases}$$

Clearly, if we have a stream of blocks x we can also calculate a stream of checksums C:

$$C = H^*(x) = [H(x^0), H(x^1), H(x^2), \ldots, H(x^n)] = [c^0, c^1, c^2, \ldots, c^n]$$

So, given a stream of blocks x and a stream of corresponding checksums C, a stream is valid if:

$$\forall (x^i \in X)\ \exists! (c^i \in C) : H(x^i, c^i)$$

And a stream is corrupt if:

$$\exists (x^i \in X)\ \exists! (c^i \in C) : \neg H(x^i, c^i)$$

Where i denotes the index in the streams X and C.

4.2.4 Stream Splitting & Merging

Consider that we have a stream x which consists out of n blocks. Blocks can be of size λ or μ as defined earlier. And that we want to split x into m streams x' in such a way that we require at least f streams, where $f \leq m$, to regain the original stream x. Each stream x' will consist of w blocks of size γ, thus:

$$x'_m = [x'^0_m, x'^1_m, x'^2_m, \ldots, x'^w_m]$$

Each block x^i in x is split in to m blocks x'^i by function db:

$$db(x^i, m, f) = [x'^i_0, x'^i_1, x'^i_2, \ldots, x'^i_m]$$

Thus, the function $d(x, m, f)$:

$$d(x, m, f) = \begin{bmatrix} [x'^0_0, x'^1_0, x'^2_0, \ldots, x'^w_0] \\ [x'^0_1, x'^1_1, x'^2_1, \ldots, x'^w_1] \\ [x'^0_2, x'^1_2, x'^2_2, \ldots, x'^w_2] \\ \vdots \\ [x'^0_m, x'^1_m, x'^2_m, \ldots, x'^w_m] \end{bmatrix}$$

Merging is possible with at least f non-corrupt streams, thus as long as the number of corrupt streams $b \leq (m - f)$ merging is possible. The composition function c is trivial:

$$c(x'^0, x'^1, x'^2, \ldots, x'^f) = x$$

c will compose each block x^i using function cb:

$$cb(x'^i_0, x'^i_1, x'^i_2, \ldots, x^i_f) = x^i$$

We assume that the composition function has the following property:

$$\begin{aligned} cb(x'^i_0, x'^i_1, x'^i_2, \ldots x'^i_f) &= x^i \\ cb(x'^i_0, x'^i_1, x'^i_2, \ldots x'^i_f) &\neq cb(x'^i_1, x'^i_0, x'^i_2, \ldots x'^i_f) \\ cb(x'^i_0, x'^i_1, x'^i_2, \ldots x'^i_f) &\neq cb(x'^i_2, x'^i_1, x'^i_0, \ldots x'^i_f) \\ &\vdots \\ cb(x'^i_0, x'^i_1, x'^i_2, \ldots x'^i_f) &\neq cb(x'^i_f, x'^i_{f-1}, x'^i_{f-2}, \ldots x'^i_0) \end{aligned}$$

The value of w is depended on the actual implementation of the decomposition method, but in general we assume that $w \leq n$. Also, we make the same assumptions

about the size of blocks in x' as in x, thus all but the last block in x' are size γ.

4.3 Streaming DepSky-A

Streaming DepSky-A is the first and most basic variant of our two algorithms. It improves the availability and integrity of cloud-stored data by replicating it on several providers using quorum techniques. This leads to a space efficiency of $\frac{1}{n}$, where n is the number of clouds.

The calculated checksums are stored in a metadata file along side the actual file. The function $write\_metadata\_a(du, M)$ will write the metadata file M to a quorum of $n - f$ clouds. The function $read\_metadata\_a(du)$ will read the correctly signed metadata for file du from $n-f$ out of n clouds and return one. f denotes the number of faulty clouds.

The function $write\_cloud(i, du, b, \beta)$ will write block β for the file du to cloud i at position b. This function will write all blocks β for file du to cloud i ordered by increasing b, or it will fail with an error if a block can not be written.

The functions db and cb introduced in Section 4.2 (page 20) are different for each algorithm. Therefore, we will further specify how these functions behave for DepSky-A.

The block decomposition function db is defined as:

$$db(x^i, m, f) = [x_0^i, x_1^i, x_2^i, \ldots, x_m^i]$$

Note that:

$$x'^i = x^i$$

Thus the size of each decomposed block is the same as the original block:

$$\gamma = |x^i| = |x'^i|$$

4.3.1 Pseudocode

The write algorithm is shown in Algorithm 1. The first step is to check whether the file already exists. If so then we return an error, or throw an exception[22], as we do not support file versioning (lines 2 – 4). If the file does not exist, we split the stream into blocks and create the decomposed streams (lines 5 – 7). We also create an empty metadata file that can be used to store the metadata for each decomposed stream (line 8).

[22]Implementation specific.

These decomposed streams will be uploaded in parallel to n cloud providers. Each block (denoted by β) is checksummed by the cryptographic hash function H, the checksum is stored in the collection $M.h$, and the block is uploaded to the cloud provider (lines 10 – 15). Finally, when all blocks are processed, the counter d is incremented atomically to indicate a successful upload (line 16). Once we receive $n - f$ responses (thus $d \geq (n - f)$), the algorithm continues.

The last step is to create an unforgeable signature for the metadata file, and upload the metadata to each cloud to terminate the algorithm (lines 19 – 20). The act of first uploading the data and then the metadata ensures that if metadata for a file is retrievable, then the data will also be available as it has already been written to $n - f$ clouds.

The read algorithm in Algorithm 2 will first download the metadata, and check if the file exists. If the file does not exist, an error is raised and the algorithm terminates. Otherwise we allocate counters for the number of blocks processed and the total number of blocks. Then using a for loop we retrieve all the blocks. Each iteration of the loop will start a parallel loop to retrieve the value of the current block c from the clouds. Since a valid block is available on at least $n - f - f$ clouds, each request for a block will succeed and thus B_c will be set. This is done using an atomic compare and swap method to ensure that the value is only set once. An optimization would be to skip the atomic compare and swap and simply set the value (atomically!), as each retrieved block tmp_i will result in the same B_c. The block is then placed at the proper position in the result stream x and all pending requests for that block are cancelled. When all blocks are retrieved x is returned.

Algorithm 1 Streaming DepSky-A (Write)

```
1:  procedure S-DepSky-A-Write(K_pr, du, x)
2:      if read_metadata_a(du) ≠ ∅ then
3:          return File exists error
4:      end if
5:      d ← 0                                   ▷ Counter for completed requests
6:      x^b ← r(x, λ)                           ▷ Convert stream x to stream of blocks
7:      X ← d(x^b, n, f + 1)                    ▷ Decompose stream
8:      M ← empty metadata
9:      parallel for i = 0 → n do
10:         x' ← X_i
11:         for b = 0 → |x'| do
12:             β ← x'^b
13:             M.h[b] ← H(β)                   ▷ Atomic operation
14:             write_cloud(i, du, b, β)
15:         end for
16:         d ← d + 1                           ▷ Atomic increment
17:     end parallel for
18:     wait until d ≥ (n − f)
19:     sign(M, K_pr)
20:     write_metadata_a(du, M)
21: end procedure
```

Algorithm 2 Streaming DepSky-A (Read)

```
1:  function S-DepSky-A-Read(du)
2:      M ← read_metadata_a(du)
3:      if M = ∅ then
4:          return File does not exist
5:      end if
6:      c ← 0                                   ▷ Number of blocks processed.
7:      t ← |M.h|                               ▷ Number of blocks in stream.
8:      x ← [ ]                                 ▷ x is the result stream, initialized to empty.
9:      for c → t do
10:         B_c ← ∅
11:         parallel for i = 0 → n do
12:             tmp_i ← block c from cloud i
13:             if H(tmp_i, M.h[c]) then
14:                 cmp_set(B_c, ∅, tmp_i)      ▷ Atomic compare and set.
15:             end if
16:         end parallel for
17:         wait until B_c ≠ ∅
18:         x_c = cb(B_c)                       ▷ Set position c in x to the composed block.
19:         Cancel pending requests for block B_c
20:     end for
21:     return x
22: end function
```

4.3.2 Proof of correctness

We observe that a correct process will not block executing $read\_metadata\_a$ or $write\_metadata\_a$ by Lemma 1.

LEMMA 1. *A correct process will not block when executing* $read\_metadata\_a$ *or* $write\_metadata\_a$.

Proof. $write\_metadata\_a$ requires $n - f$ responses, these responses are simple *acks* and will always be received since we are using reliable communication channels and at most f cloud providers will be faulty. $read\_metadata\_a$ will read a valid metadata file from a quorum of cloud providers. Since we are considering only non-malicious writers, a metadata file written in a cloud is always valid and correctly signed using K_{pr}. Thus, valid metadata will be read from $n - f$ clouds, from which one can arbitrarily be selected to finish the algorithm. □

Because we first upload the data, and only when that is written to $n - f$ clouds, we write the metadata, we can state that the value associated with the metadata returned by $read\_metadata\_a$ is available on at least $f + 1$ non-faulty-clouds. We formalize this in Lemma 2 [1].

LEMMA 2. *The value associated with the metadata returned by* $read\_metadata\_a$ *is available on at least* $f + 1$ *non-faulty clouds.*

Proof. Recall that only valid metadata files will be retrieved by $read\_metadata\_a$. These metadata files will be written only by a non-malicious writer that follows the streaming DepSky protocols. The data files will be written to a quorum of $n - f$ clouds, and then the metadata is written to a quorum of clouds. Consequently, a metadata file is only put on a cloud if its associated value was already put on a quorum of clouds. It implies that if a metadata is read, its associated value was already written to $n - f$ servers, from which atleast $n - f - f \geq f + 1$ are correct. □

The write algorithm is wait-free by Theorem 1.

THEOREM 1. *The Streaming DepSky-A write algorithm is a wait-free operation.*

Proof. By Lemma 1 we know that $read\_metadata\_a$ and $write\_metadata\_a$ will not block. The loop will upload the file to n cloud providers in parallel, each iteration will result in an *ack* from the cloud provider or a failure. For each successful upload of a complete stream, d will be increment atomically. Since at most f stream uploads will fail, the wait condition $d \geq (n - f)$ can always be satisfied. Thus releasing the barrier and completing the algorithm. □

The read algorithm is wait-free by Theorem 2.

THEOREM 2. *The Streaming DepSky-A read algorithm is a wait-free operation.*

Proof. By Lemma 1 we know that $read\_metadata\_a$ will not block. By Lemma 2 we know that if the metadata is retrievable, then the data will be stored on atleast $f+1$ clouds. Thus, atleast $f+1$ iterations of the retrieval loop will do an atomic compare and set for each block. Thus, each block will be retrievable, completing the for loop and allowing the algorithm to complete. □

4.4 Streaming DepSky-CA

While Streaming DepSky-A already improves the availability and integrity of cloud-stored data, it does not guarantee the confidentiality of the data. Therefore we introduce our second algorithm: Streaming DepSky-CA. This algorithm uses a symmetric encryption algorithm and a secret sharing scheme, to ensure that only readers with access to $f+1$ cloud providers can read the data.

We also optimize the space efficiency of the storage mechanism using a *information-optimal erasure code*. This allows us to improve the space efficiency to:

$$\frac{1}{\frac{n}{f+1}} = \left(\frac{n}{f+1}\right)^{-1} = \frac{f+1}{n}$$

Especially when f (the number of faulty clouds) increases (and consequently n as $n = 3f+1$) this will lead to significantly more cost efficient storage (plotted in Figure 3 (page 27)):

$$\lim_{n\to\infty} \frac{1}{n} = 0 < \lim_{f\to\infty} \left(\frac{f+1}{n} = \frac{f+1}{3f+1}\right) = \frac{1}{3}.$$

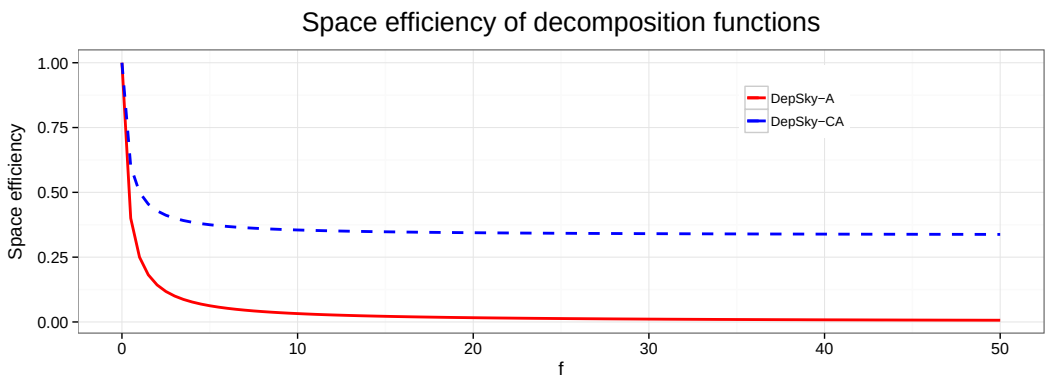


Figure 3: Space efficiency as a function of f

To ensure confidentiality of stored data we utilize encryption, and to maintain acceptable performance we will use a symmetric encryption algorithm. The encryption

function enc accepts a stream x and a secret key ϕ to generate an encrypted stream x^c:

$$x^c = enc(x, \phi)$$

The decryption function dec accepts a (encrypted) stream x^c and the secret key ϕ to generate the decrypted stream x:

$$x = dec(x^c, \phi)$$

Which satisfies:

$$x = dec(enc(x, \phi), \phi)$$

To generate the key used by the encryption function we introduce the function gen, which takes no arguments and outputs a random key usable by enc and dec:

$$\phi = gen()$$

To share the secret keys using by the encryption algorithm with the readers, but not with the cloud providers, we will use a *secret sharing scheme*. Specifically a (k, n) threshold scheme, such as the one proposed by Shamir in [26]. This gives us the function sss:

$$sss(D, n, k) = [D_0, D_1, D_2, \ldots, D_n]$$

Which will split a secret D into n pieces so that the knowledge of k or more pieces D_i makes D easily computable, and the knowledge of $k-1$ or fewer D_i pieces leaves D completely undetermined[23]. If $k = n$ then all shares are required to reconstruct D.

Obviously we also require the inverse function of sss to combine the split secret again, thus:

$$sss^{-1}(D_0, D_1, D_2, \ldots, D_i) = D$$

where $k \leq i \leq n$, which is no different from the original DepSky-CA [1].

4.4.1 Important changes from Streaming DepSky-A

The Streaming DepSky-CA algorithm is a modified version of the Streaming DepSky-A algorithm. Metadata is retrieved using the $read\_metadata\_ca$ function which returns a set of $n - f$ metadata files. The $write\_metadata\_cloud$ function is used to write metadat to the clouds, it will write a metadata file to a single cloud.

An important difference between Streaming DepSky-A and Streaming DepSky-CA is

[23] This means that all possible values are equally likely.

the data stored in the metadata. Each metadata file contains the hashes for all cloud providers. The hashes for each cloud provider will be different due to the conceptual processing order for each block:

1. Read block
2. Encrypt block
3. Erasure encode block
4. Hash block
5. Store block

Obviously, we will have to reverse these steps when reading the data. The erasure coding will encode a block of data in such a way that we require $f + 1$ clouds to decode the block again. If we do not store the block hashes for each block, for each cloud, in the metadata for all clouds we can not guarantee this.

To illustrate this see Figure 4 (page 29). In this figure we show the situation when we do not store the metadata for all clouds in all the metadata files. We assume the worst case, thus that f cloud providers will fail every request, and that not the same f providers will fail every time. We note that, theoretically, we require an additional $2f + 1$ non-faulty providers to overcome f faulty ones by [24].

We see that the data and the metadata are stored in $n - f$ clouds, but they are stored in different quorums. Note that the metadata for the failed write will be incomplete, and is thus also invalid even though it might be stored properly. This leaves $(3f + 1) - 2f = f + 1$ cloud providers to retrieve the file, which is the bare minimum we require. If we would occur another f failures, the data is irretrievable.

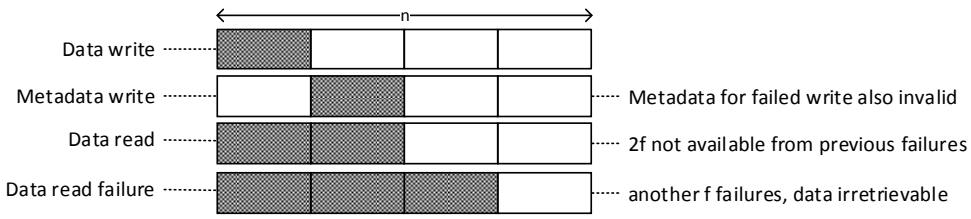


Figure 4: Graphical illustration of metadata hash storage argument, failure case.

Clearly, this is not an ideal situation. There are two solutions to this, the first is to hash the data before it is encoded, or to store the hashes for all clouds in all metadata files. This is a classical time-space tradeoff: do we want to save computational effort? Or do we want to improve computational efficiency at the cost of using more memory and storage space? We chose the latter because we aim at lowering the cost of processing a single block of data, thus increasing the potential throughput of the

system. We reason that a CPU bound task is much harder to improve than a task that requires more memory. Also, adding more memory to a system that runs the task is much cheaper than adding more processing units.

Figure 5 (page 30) shows the result when the hashes for all clouds are available from the metadata. This brings the Streaming DepSky-CA algorithm back in line with Lemma 2, which states that if metadata is retrievable, that also the data will be retrievable from atleast $f + 1$ non-faulty clouds.

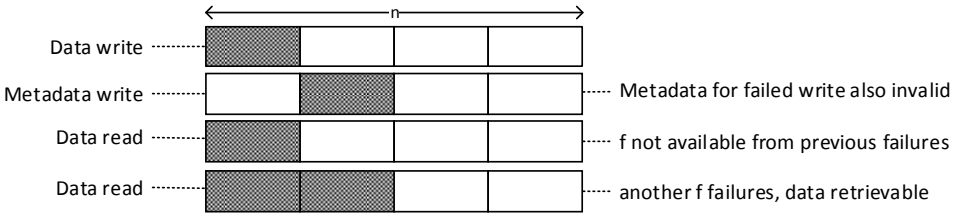


Figure 5: Graphical illustration of metadata hash storage argument, success case.

The metadata size will be:

$$s = p + \sum_{x' \in X} \sum_{\beta \in x'} |H(\beta)|$$

Where p is a constant factor of the metadata (such as the shared secret, signature, etc.) and X is the set of erasure encoded streams.

4.4.2 Pseudocode

The changes to the pseudocode for Streaming DepSky-CA (Algorithm 3) are minor. We added the generation of a secret key (line 6), the encryption of the stream (line 7), and the sharing of the secret (lines 25-33) to satisfy our confidentiality requirement. Hashes are stored in the metadata hash collection, of which we now have n (line 17). Also the writing of the metadata to the clouds has been in lined.

The read algorithm in Algorithm 4 has only minor changes. The secret key is first combined from the metadata (line 6), and the key is used the return the decrypted stream at the end (line 22). Then we determine the amount of blocks we will have to read by taking the maximum number of hashes from all n cloud providers. Since writers can only fail by crashing, the maximum amount of hashes will always be the number of blocks written to valid clouds. The reading from the clouds will now require at least $f + 1$ blocks to be read, as opposed to a single one in Streaming DepSky-A (line 18). These are then composed and stored in the result stream (line 19).

4.4.3 Proof of correctness

The proof of correctness for the Streaming DepSky-CA algorithm is quite similar to the proof of Streaming DepSky-A. The $read\_metadata\_ca$ function adheres to a modified version of Lemma 1, Lemma 3.

LEMMA 3. *A correct process will not block during the execution of* $read\_metadata\_ca$.

Proof. The $read\_metadata\_ca$ function will return a set of $n - f$ valid metadata. Since we are considering only non-malicious writers, a metadata file written in a cloud is always valid and correctly signed using K_{pr}. The use of reliable communication channels will guarantee that valid metadata will be read from $n - f$ clouds, which will be returned as an array. □

Since $read\_metadata\_ca$ will return an array of $n - f$ metadata, and $n - f \geq f + 1$, we will always be able to recover the secret key ϕ to decrypt the file as this only requires $f + 1$ shares.

As the Streaming DepSky-A algorithms, the Streaming DepSky-CA algorithms are also wait-free, by Theorem 3 and Theorem 4.

THEOREM 3. *The Streaming DepSky-CA write algorithm is a wait-free operation.*

Proof. By Lemma 3 we know that $read\_metadata\_ca$ will not block. The loop to upload the data will execute n parallel iterations of which at most f will fail, thus the lock condition $d \geq (n - f)$ can always be satisfied. The same holds for the uploading of the metadata: at most f iterations of the metadata upload parallel loop will fail. Thus, we can also always satisfy the lock condition $m \geq (n - f)$ and terminating the algorithm. □

THEOREM 4. *The Streaming DepSky-CA read algorithm is a wait free operation.*

Proof. By Lemma 3 we know that $read\_metadata\_ca$ will not block. We also know that $read\_metadata\_ca$ will return a set of $n-f \geq f+1$ metadata files, thus allowing us to reconstruct the secret key ϕ. Since by Lemma 2 we know that a file for which metadata is retrievable is also available on at least $f + 1$ clouds, the lock condition for each block $|B_c| \geq f + 1$ can always be satisfied. Logically it follows that then each block of the file will be retrievable, terminating the loop. Since the decryption key is also available, the correctly decrypted stream can be returned after which the algorithm terminates. □

Algorithm 3 Streaming DepSky-CA (Write)

```
1:  procedure S-DepSky-CA-Write(K_pr, du, x)
2:      if read_metadata_ca(du) ≠ ∅ then
3:          return File exists error
4:      end if
5:      d ← 0                                   ▷ Counter for completed requests
6:      φ ← gen()                               ▷ Secret key
7:      x^c ← enc(x, φ)                         ▷ Encrypted stream
8:      x^b ← r(x^c, λ)                         ▷ Convert stream x^c to stream of blocks
9:      X ← d(x^b, n, f + 1)                    ▷ Decompose stream
10:     M ← empty metadata

11:     i ← 0
12:     parallel for i → n do
13:         x' ← X_i
14:         b ← 0
15:         for b → |x'| do
16:             β ← x'^b
17:             M.h_i[b] ← H(β)                 ▷ Atomic operation
18:             write_cloud(i, du, b, β)
19:         end for
20:         d ← d + 1                           ▷ Atomic operation
21:     end parallel for
22:     wait until d ≥ (n − f)

23:     m ← 0
24:     i ← 0
25:     S ← sss(φ, n, f + 1)                    ▷ Distribute key shares
26:     parallel for i → n do
27:         M_l ← M                             ▷ Create local copy of M
28:         M_l.key ← S_i
29:         sign(M_l, K_pr)
30:         write_metadata_cloud(du, i, M_l)
31:         m ← m + 1                           ▷ Atomic operation
32:     end parallel for
33:     wait until m ≥ (n − f)
34: end procedure
```

Algorithm 4 Streaming DepSky-CA (Read)

```
1:  function S-DepSky-CA-Read(du)
2:      M ← read_metadata_ca(du)
3:      if M = ∅ then
4:          return File does not exist
5:      end if
6:      φ ← sss^{-1}(M_0.key, M_1.key, ..., M_{n-f}.key)
7:      c ← 0
8:      t ← max(|M_0.h_0|, |M_0.h_1|, ..., |M_0.h_n|)          ▷ M_0 is arbitrary
9:      x ← [ ]                                  ▷ Result stream, initialized to empty.
10:     for c → t do
11:         B_c ← ∅
12:         parallel for i = 0 → n do
13:             tmp_i ← block c from cloud i
14:             if H(tmp_i, M.h_i[c]) then
15:                 B_c ← B_c + tmp_i                          ▷ Atomic operation
16:             end if
17:         end parallel for
18:         wait until |B_c| ≥ f + 1
19:         x_c = cb(B_c^0, B_c^1, ..., B_c^{f+1})   ▷ Set position c in x to the composed block.
20:         Cancel pending requests for block B_c
21:     end for
22:     return dec(x, φ)
23: end function
```

5 Implementation

The functions and algorithms introduced in Chapter 4 were also implemented in a proof of concept application. This chapter will link the fundamental functions from Chapter 4 to concrete components in a software architecture. The implementation was written in the Java programming language as requested by TNO. The **J**ava **D**evelopment **K**it (JDK) used was the Oracle JDKDK version 7.

5.1 Basic Components

Since dealing with streams of data is a common practice, the JDK supplies the most fundamental component, the `InputStream` abstract class, out of the box. This class allows data to be read one byte at a time, or into preallocated `byte` arrays. The creation and verification of unforgeable signatures is also part of the JDK, provided by implementations of the `Signature` class. The calculation of cryptographic hashes is also provided by the JDK `MessageDigest` class. Implementations `MessageDigest` class are provided to allow the calculation and verification of commonly used algorithms, such as MD5 and the Secure Hash Algorithm (SHA) family. We will still wrap these components to create a more specialized and easy to use interface for components that require these functions. In Figure 6 the UML diagrams of these classes are shown.

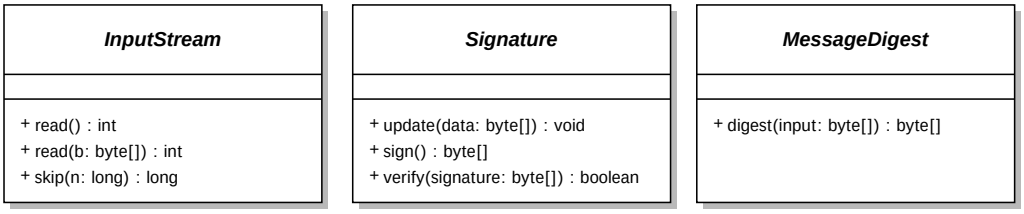


Figure 6: Basic components provided by the JDK

5.1.1 Stream reading

The `InputStream` read method does not guarantee that the complete block will be read from the stream, instead it only guarantees that at least 1 byte will be read. Most common implementations read until the next byte that would be read causes a blocking operation, or the end of the stream is reached. Therefore we encapsulate the `InputStream` in a `BlockReader` object. Note that by design the `InputStream` will return the special value -1 when it reaches the end of the stream[24].

[24]This is ofcourse a 'magic' constant and should have been stored in a constant for comparison.

The `internalRead` function (Listing 1) handles the reading of full blocks. This is a blocking operation until the `InputStream` returns enough bytes, or the end of the stream is reached. During the while loop bytes will be added to the internal buffer until we have `blockSize()` bytes, or until the end of stream value is returned by the `InputStream`. The `internalRead` function is called by the `read()` and `read(ByteBuffer)` public methods, which return the data read into the internal buffer encapsulated in a new read-only `ByteBuffer`, or copied into the given `ByteBuffer` respectively.

Listing 1: BlockReader.internalRead function

```
private int internalRead() throws IOException {
  if ( endOfStream)
    return -1;

   checkClosed(); // Throws exception if closed.

  /* Will read into internalBuffer, starting at offset 0, at most
      blockSize() bytes.
   * Returns amount of bytes read.
   */
  int bytesRead = in. read( internalBuffer, 0, blockSize());

  if ( bytesRead == -1) {
     endOfStream = true;
    return -1;
  }

  while ( bytesRead != blockSize()) {
    int bytesReadThisPass = in. read( internalBuffer, bytesRead,
                    ( blockSize() - bytesRead));

    if ( bytesReadThisPass == -1)
      break;

     bytesRead += bytesReadThisPass;
  }

  return bytesRead;
}
```

Since we want to allow users to program against interfaces and not concrete implementations, we introduced the `IBlockReader` interface which exposes the functions of the `BlockReader`. Since we also required a Factory to create a `IBlockReader` for each `InputStream` we have to process, we also required a `IBlockReaderFactory`. This lead to the use of the Abstract Factory Pattern[25], which lead to the class layout as shown in Figure 7 (page 36).

[25]Explained in more detail in Appendix A4.1.

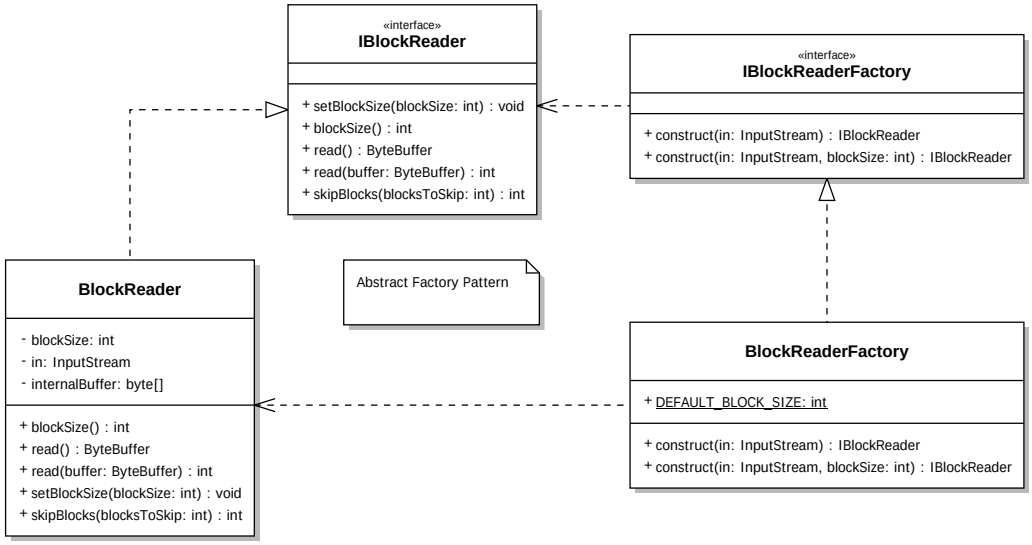


Figure 7: Class Diagram for the BlockReader showing the Abstract Factory Pattern

Verification We will now verify if our implementation matches the function r we introduced in Chapter 4.

The definition of the block reading function is

$$x' = r(x, \lambda) = [\beta_0, \beta_1, \beta_2, \ldots, \delta]$$

and we note that

$$\lambda = |\beta|$$

and

$$\mu = |\delta| = |x| \bmod \lambda.$$

The value returned by the function `blockSize()` is configurable and corresponds to λ. Each block that is read by `internalRead` is thus of size λ or the remainder of the stream $|x| \bmod \lambda \Longrightarrow \mu$. Thus, we can conclude that the function r is correctly implemented by the `BlockReader` class. The `BlockReader` instance itself is the result stream x'.

5.1.2 Unforgeable Signatures

The unforgeable signatures provided by the `Signature` class work (simplified) by calculating a cryptographic hash of the data, and then encrypting that hash with the private key. The verification is done by calculating a new hash of the data, and decrypting the encrypted hash using the public key. If the newly calculated hash and the decrypted hash match, then we can assume that the data has not been tampered with. Obviously if the private key is cracked, stolen, or otherwise acquired by the

attacker this assumption is no longer valid.

To increase the amount of effort the attacker has to do to crack the private key, increasing the size of the key (measured in bits) is the easiest way. RSA (the company, not the algorithm) recommends a minimum key size of 2048 bits for data that should be confidential up to 2030. For data that should remain confidential after 2030 they recommend a key size of 3072 bits [27]. The authors of [28] are a bit more conservative than the RSA and recommend a key size of at least 2048 bits, but if it can be afforded 4096 bits (or as close as possible to this number). They also recommended that the software supports 8192 bit keys "just in case". We will follow recommendation of [28] to use a key size of 4096 bits, which should only be lowered if the decryption of the metadata signature proves to be a significant bottleneck.

To ease the management of the public and private keys, as well as making the `Signature` class more user friendly, we introduce the `IMetadataSignatureStrategy` interface. As the name suggests, the Strategy pattern[26] is applied to allow it to be replaced easily by other implementations. Out of the box an implemention with the RSA algorithm and a SHA-512 hash is provided.

Listing 2: Implementation of signing using the Signature class

```
1  public byte[]  sign(final byte[]  contents) {
2    if ( privateKey == null)
3      throw new  KeyNotAvailableException("privateKey");
4
5    // Exception handling removed for clarity.
6       signatureAlgorithm. initSign( privateKey); // State!
7       signatureAlgorithm. update( contents);
8      return  signatureAlgorithm. sign();
9    // Exception handling removed for clarity.
10 }
```

Since the `Signature` implementation is stateful (see Listing 2, lines 6 & 7), an invocation to sign or verify data is not thread safe. This means that a single instance can not be shared between threads without appropriate synchronization. Since we don't know how much congestion there will be on these functions, we applied the Abstract Factory pattern so that we can construct a new instance each time it is required. Which results in the class layout given in Figure 8 (page 38).

If congestion turns out not to be a significant bottleneck, then it is possible to remove the Abstract Factory pattern and provide a single, properly synchronized, instance of a `IMetadataSignatureStrategy` implementation to the reader and writer (or one instance for each).

[26]The Strategy pattern is explained in more detail in Appendix A4.2.

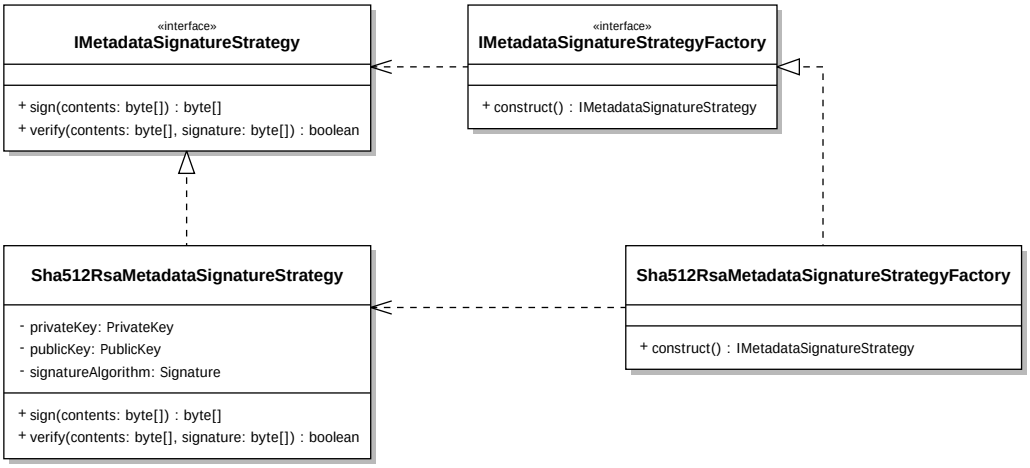


Figure 8: Metadata signature class layout

If the creation of instances adds significant processing overhead, then the application of the Object Pool Pattern[27] inside the `IMetadataSignatureStrategyFactory` can be considered. This requires asserting that the reader and writer properly acquire and release instances of `IMetadataSignatureStrategy` implementations. Because instances can now be reused, this reduces the overhead because less instances have to be created. This is a time-memory trade off, as the instances in the pool will consume memory.

Verification The fundamental functions for signing and verifying unforgeable signatures were defined in Chapter 4 as:

$$\sigma = sign(data, K_{pr})$$

and

$$verify(data, \sigma, K_{pu}) = \begin{cases} \top & \text{if the signature is valid} \\ \bot & \text{if the signature is invalid} \end{cases}$$

From Figure 8 (page 38) we can see that the interface provides two methods: `sign` and `verify`. Unlike the fundamental functions, these do not accept the key directly.

Instead the keys are stored in the (private) fields `privateKey` and `publicKey`. This gives

$$\sigma = sign'(data) = sign(data, K_{pr})$$

[27]Described in Appendix A4.3.

and

$$verify'(data, \sigma) = verify(data, \sigma, K_{pu}) = \begin{cases} \top & \text{if the signature is valid} \\ \bot & \text{if the signature is invalid} \end{cases}$$

where K_{pr} and K_{pu} are provided by the instance itself.

Since we simply delegate the actual implementation of the signing to the `Signature` implementation we satisfy the $sign$ method. If we define the bijective function $f : X \to Y$ where $\top \to$ `true` and $\bot \to$ `false`. We see that the actual implementation of $verify$ is

$$f(verify'(data, \sigma))$$

We conclude that the implementation matches the definition.

5.1.3 Checksum Calculation

The checksums calculated by the `MessageDigest` implementations have to be stored in the metadata. However, we want to avoid having a lot of classes that depend on certain functionality of the metadata. Therefore, to ease the management of the cryptographic hashes calculated for each block and each cloud, we introduce the `IBlockChecksumCalculator` interface. The class layout is shown in Figure 9 (page 39).

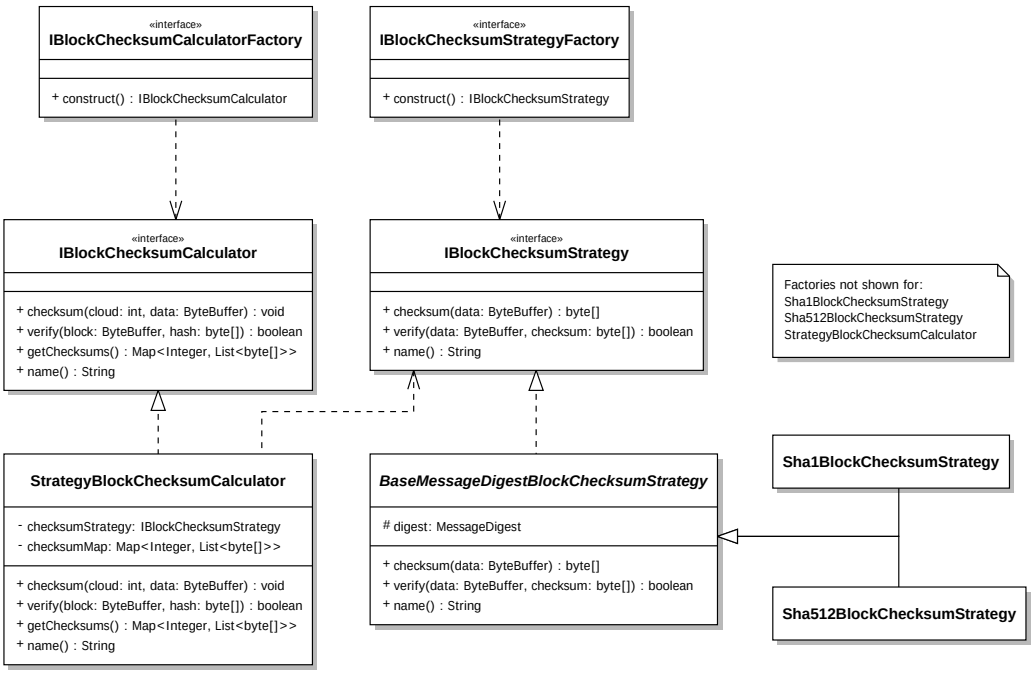


Figure 9: Class layout for checksum calculation and verification

The default implementation for the `IBlockChecksumCalculator` interface is the `StrategyBlockChecksumCalculator` class. Which, as the name suggests, can be specialized to calculate certain checksums by supplying it with an implementation of the `IBlockChecksumStrategy` interface. For the `IBlokChecksumStrategy` interface we supply a base class which can make any checksum algorithm provided by the `MessageDigest` class available. Also supplied are the implementations for two popular hash algorithms from the Secure Hash Algorithm (SHA) family, namely SHA-1 and SHA-512.

The `checksum` function accepts a cloud identifier (integer) and a `ByteBuffer` of data. The calculated checksum is added internally to a Dictionary (exposed in Java as the Map interface) which maps a cloud identifier to a list of checksums. This allows the algorithms to get all the checksums for all the clouds easily when the data has been uploaded and the metadata has to be generated. Clearly, this makes the `IBlockChecksumCalculator` a stateful object, which can not be shared between threads. Thus, the Abstract Factory pattern was also applied here.

Arguably, the `IBlockChecksumStrategy` implementations can be shared between `IBlockChecksumCalculators`. Obviously proper thread synchronization must be done to avoid corrupting the internal state of the `MessageDigest` instance, which is not guaranteed to be thread safe. However, to avoid thread congestion on the checksum calculation we applied the Abstract Factory pattern. Clearly, this also allows a consumer to easily create an Abstract Factory which provides a Singleton `IBlockChecksumStrategy`.

Verification We can clearly see that the `IBlockChecksumStrategy` interface implements both fundamental functions:

$$c = H(\beta)$$

and

$$H(\beta, c) = \begin{cases} \top & \text{if the block is valid} \\ \bot & \text{if the block is corrupt} \end{cases}$$

The actual implementation of the `IBlockChecksumCalculator` is a bit more elaborate as it also stored the checksums, something that was not defined in Chapter 4. Thus we can only conclude that the implementation of the `IBlockChecksumStrategy` is valid, provided that the `MessageDigest` implementations are valid (which is a safe assumption).

5.1.4 Stream Splitting

The implementation of the stream splitting function is the most complex fundamental component that we need. Since the splitting of the stream is done on a block by block basis, a `IBlockReader` is passed to the `StreamDecomposer`. To maintain compatibility with all the libraries and functions that have been written to handle `InputStreams`, the output of the decomposition function will be a set of `InputStream` implementations.

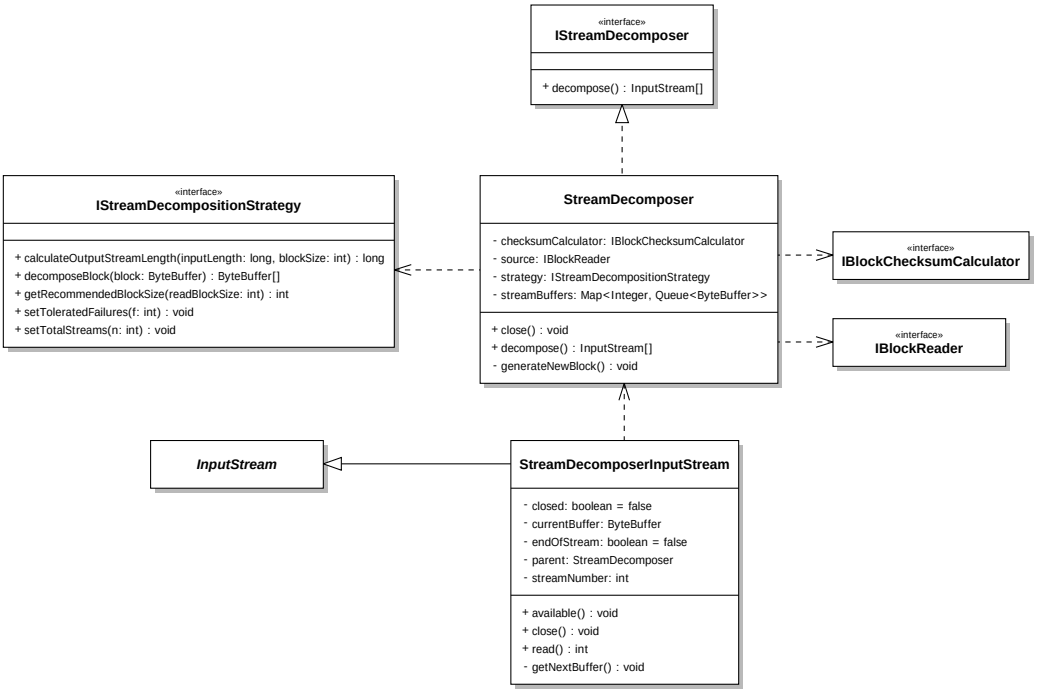


Figure 10: Class layout for Stream splitting

If we take a look at Figure 10 (page 41) we can see two components that will coordinate the splitting step: `StreamDecomposer` and `IStreamDecompositionStrategy`. The `StreamDecomposerInputStream` is a single result stream of the split. Thus, the `decompose()` function will return n instances of the `StreamDecomposerInputStream`. Each instance of `StreamDecomposerInputStream` will have a reference to its corresponding `StreamDecomposer` to retrieve its data. We will treat the way these components interact top-down, thus we will start with the function that the consumers of the `InputStream` will use: `read()`.

The activity diagram for `read` is shown in Figure 11 (page 42). Since calling `read` on a closed stream could lead to incorrect behavior, we decided to check for this and throw an exception indicating a critical failure. We do not expect that this exception will ever be thrown in production code because the closing of a stream is a clean up operation, thus calls to `read` should no longer happen. The next check is whether the end of the stream (EOS) has been reached, and return the magic constant -1 if so.

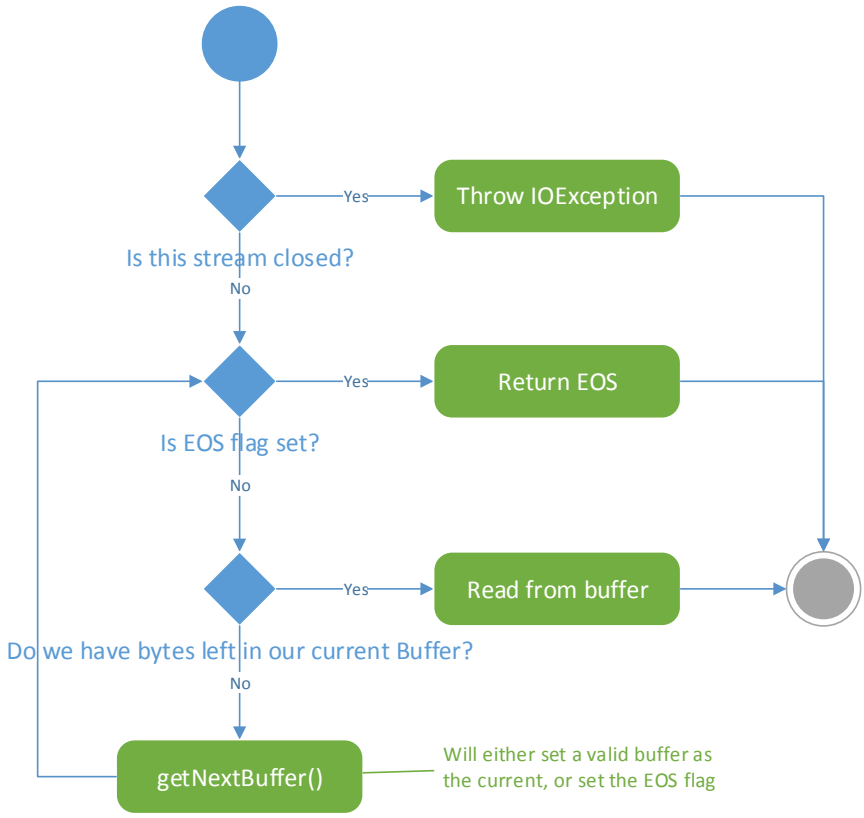
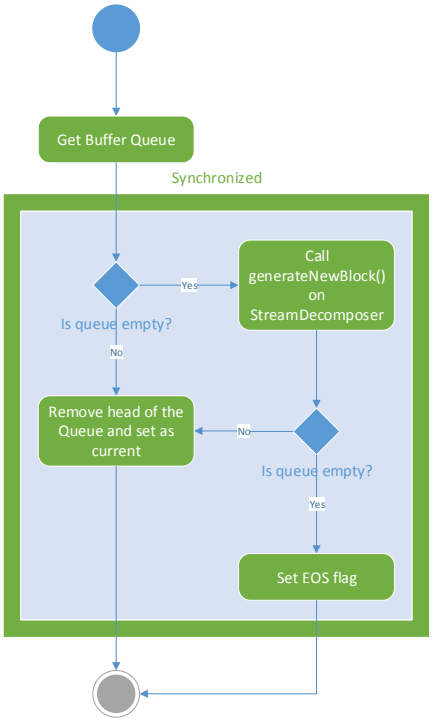


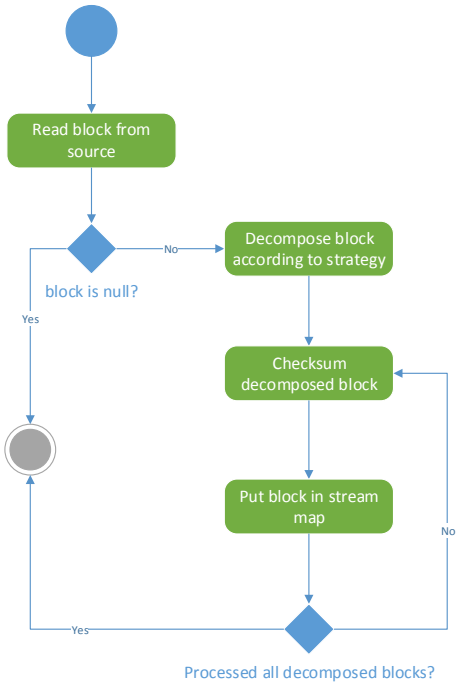
Figure 11: Activity diagram for StreamDecomposerInputStream.read

This should happen at least once in the lifetime of the object. Finally, if the current buffer has not yet been depleted, we can simply read a byte from the current buffer and return it to the caller. If there are no bytes left in the buffer, then the function `getNextBuffer` will be called.

To manage the blocks that are available for each `InputStream` the `StreamDecomposer` has a Dictionary (implemented as a HashMap) which maps a stream number (range $0 \leq i < n$) to a `Queue` of buffers. Each `InputStream` has a stream number, and a reference to its parent `StreamDecomposer` so it can retrieve the `Queue` allocated for that specific stream. Since reading from an `InputStream` is not thread safe, and we want to minimize the amount of data that is in memory at any single instance of time, we synchronized the access to the `generateNewBlock` method on the `StreamDecomposer`. This causes that threads that have depleted their current buffer simultaneously to wait for each other when retrieving the next block. This is because the check to see if the `Queue` is empty is done inside the synchronized block. Since the call to `generateNewBlock` on the `StreamDecomposer` will generate a new block for each `InputStream`, the waiting of threads should not result in a significant bottleneck. All other waiting threads will then have a block in their `Queue` and can take



(a) Activity diagram for getNextBuffer



(b) Activity diagram for generateNewBlock

the head and terminate continue. The sum of the throughput of all `InputStreams` should be roughly equal to the speed at which the original `InputStream` can be read (minus overhead introduced by the decomposition strategy and checksumming). The activity diagram for the `getNextBuffer` function is shown in Figure 12a (page 43).

In Figure 12b (page 43) the activity diagram for the final function `generateNewBlock` is shown. This function calls the `IBlockReader.read()` function, which will return a block of data or `null` as discussed earlier. Every block that is read from the stream will be passed to an implementation of the `IStreamDecompositionStrategy`, which will return an array of n blocks. On creation of the `StreamDecomposer` the `IStreamDecompositionStrategy` will be initialized with the number of blocks that should be in the result, and the number of failures we want to tolerate. The implementations are not shown here, but they refer to either the mirroring of blocks for Streaming DepSky-A or erasure coding for Streaming DepSky-CA. Each decomposed block will be submitted to the `IChecksumCalculator`, and put into the `Queues` available to the `InputStreams`.

Verification To verify if our implementation matches our design, we will reproduce the functions from chapter 4. The function d will decompose a stream x into

m streams so that we can tolerate the loss of f streams, thus:

$$d(x, m, f) = \begin{bmatrix} [x'^0_0, x'^1_0, x'^2_0, \ldots, x'^w_0] \\ [x'^0_1, x'^1_1, x'^2_1, \ldots, x'^w_1] \\ [x'^0_2, x'^1_2, x'^2_2, \ldots, x'^w_2] \\ \vdots \\ [x'^0_m, x'^1_m, x'^2_m, \ldots, x'^w_m] \end{bmatrix}$$

We see that our `StreamDecomposer` will generate an array of m streams. As noted in chapter 4 the function db is important to decompose each block:

$$db(x^i, m, f) = [x'^i_0, x'^i_1, x'^i_2, \ldots, x'^i_m]$$

We see that the function db is implemented by `IStreamDecompositionStrategy`, only slightly different as the call is $db(x^i)$ and m and f have been previously configured. We also see that if the function db is correct in its encoding, that the result of d will also be correct. As we have not defined any implementations for db, we can only conclude that the implementation of the function d is correct.

Note however, that for the function d the submission to the checksum functions was not defined. This is actually already a partial implementation of the Streaming DepSky algorithms.

5.1.5 Stream Merging

To merge the streams read from the cloud providers we will use the `StreamComposer` class. When implementing the `StreamComposer` we do not have to consider that the resulting `InputStream` has to be thread safe. Thus the implementation is slightly easier than that of the `StreamDecomposer`. The class layout is shown in Figure 13 (page 45).

The `StreamComposer` will be supplied with n or $n - f$ `CompositionParts`. Each `CompositionPart` contains a reference to a `InputStream` from a cloud provider, and the set of cryptographic hashes that were calculated for the stream. An instance of the `IStreamCompositionStrategy` interface is created and configured with the amount of streams that are required to restore blocks, which the `StreamComposer` will use to determine how many streams need to be supplied to the Strategy.

The result `InputStream` can been obtained by a call to `StreamComposer.compose()`, and bytes can be read from the stream using the `read` method. Like the `read` method implemented by the `StreamDecomposerInputStreams` the `ComposedInputStream` uses a reference to the parent `StreamComposer` to obtain the next block to read. The activity diagram for the read method is shown in Figure 14 (page 46). It is very

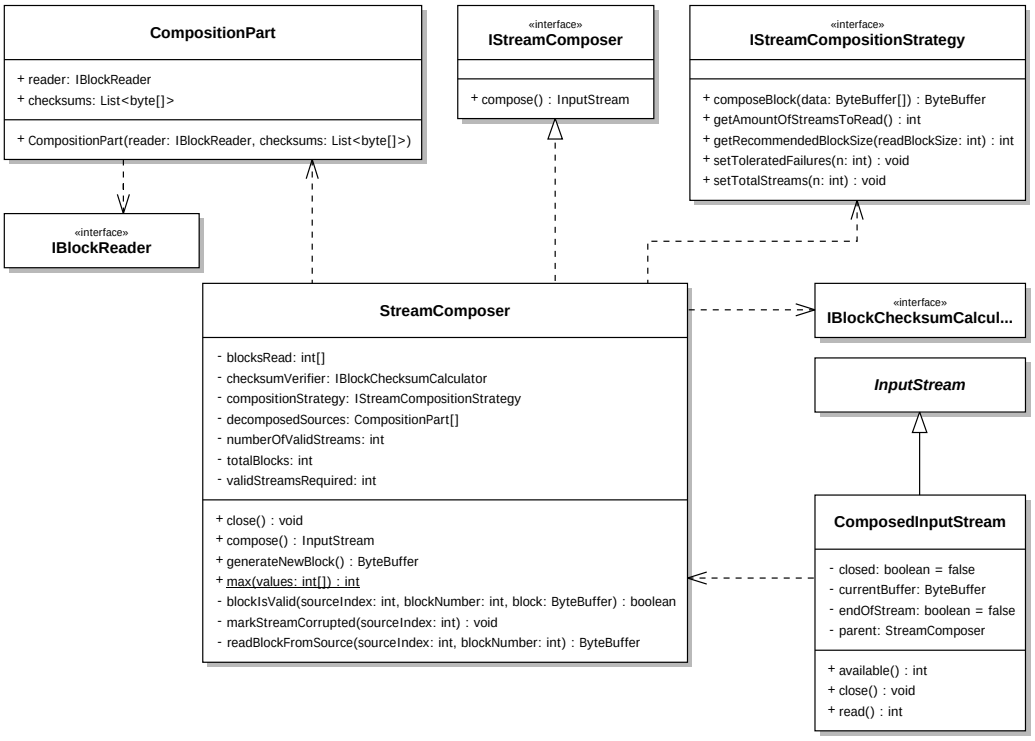


Figure 13: Class layout for StreamComposer

similar to the `read` method of the `StreamDecomposerInputStream` class. Closed streams, again, throw an exception to indicate a critical failure. Once then end of stream flag is set, the stream will continue to return -1 on invocations of the method. If these conditions do not apply, and the current buffer still has bytes remaining in it, these will be read. Once the current buffer has been depleted the `generateNewBlock` function on the parent `StreamComposer` is called. If the buffer returned by `generateNewBlock` is `null` then we set the EOS flag.

The activity diagram for the `generateNewBlock` function is shown in Figure 15 (page 47). The first check is to see if there are more blocks available in the stream, this is done using a counter. If we conclude that there are not enough streams available to do a composition, we throw an IOException. Initially all streams are considered available, and when an invalid block is read from a stream it is marked as invalid. If there should still be more blocks available in the stream, and we have enough valid streams available, a pointer is allocated which points to the reader of the first `CompositionPart`. To ease the reading of the activity diagram the steps of allocating a pointer and checking for enough valid streams have been swapped around.

If a block read from the current stream (the current stream is indicated by the pointer allocated earlier) fails the hash check, the current stream is marked as invalid and is closed. If there are more streams available, the pointer is moved and the block is read from a different stream. If the block is valid, it is stored

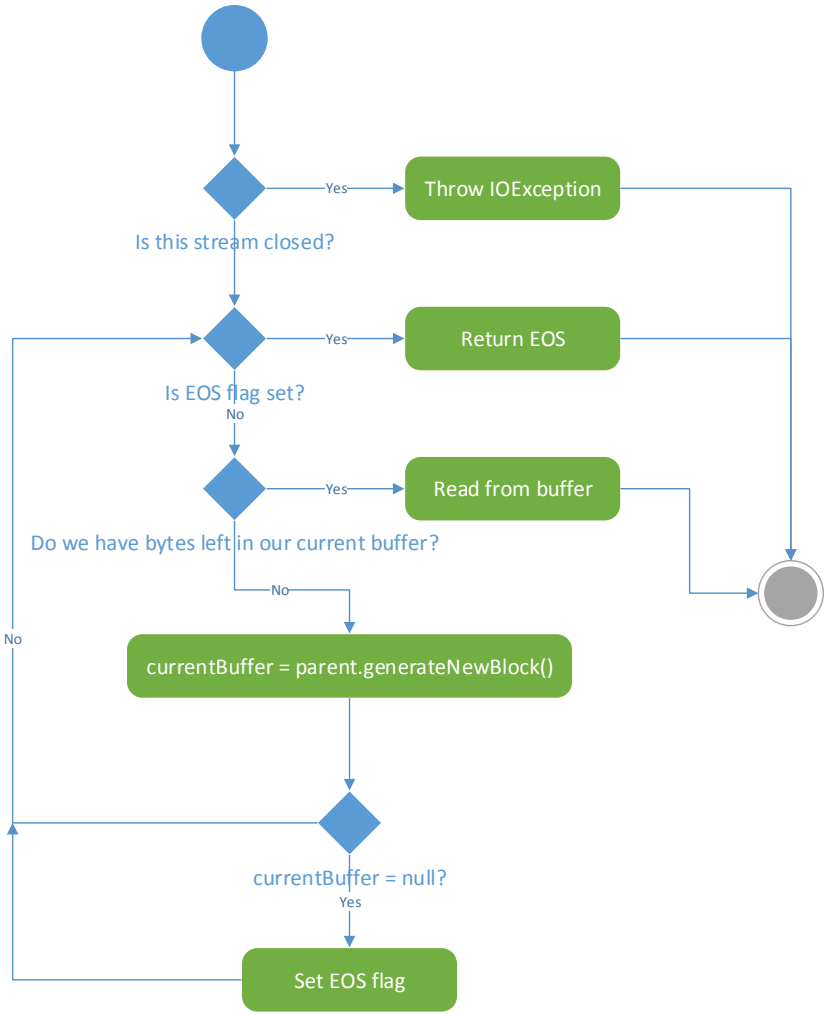


Figure 14: Activity diagram for ComposedInputStream.read

in an array. Once enough blocks are available the blocks are submitted to the `ICompositionStrategy` and the composed block is returned to the caller.

This function will always do one of three things: return a valid composed block, return null to indicate the end of the stream, or throw an `IOException` indicating an error condition. This should be clear if we analyze the function. First, if there are no more blocks available then `null` is returned. This is a success scenario. Second, if we see that there is no way that we can provide a valid block, but there should be more, then an `IOException` is thrown. This is the quick fail scenario. Further, we see that if no failures happen a successfully composed block can always be returned, this should be the default success scenario.

There two other failure scenarios that can occur, however these are slightly less obvious to see from the diagram. If we assume that we have n `CompositionParts` and require $f + 1$ blocks to compose into one valid output block, then if during the

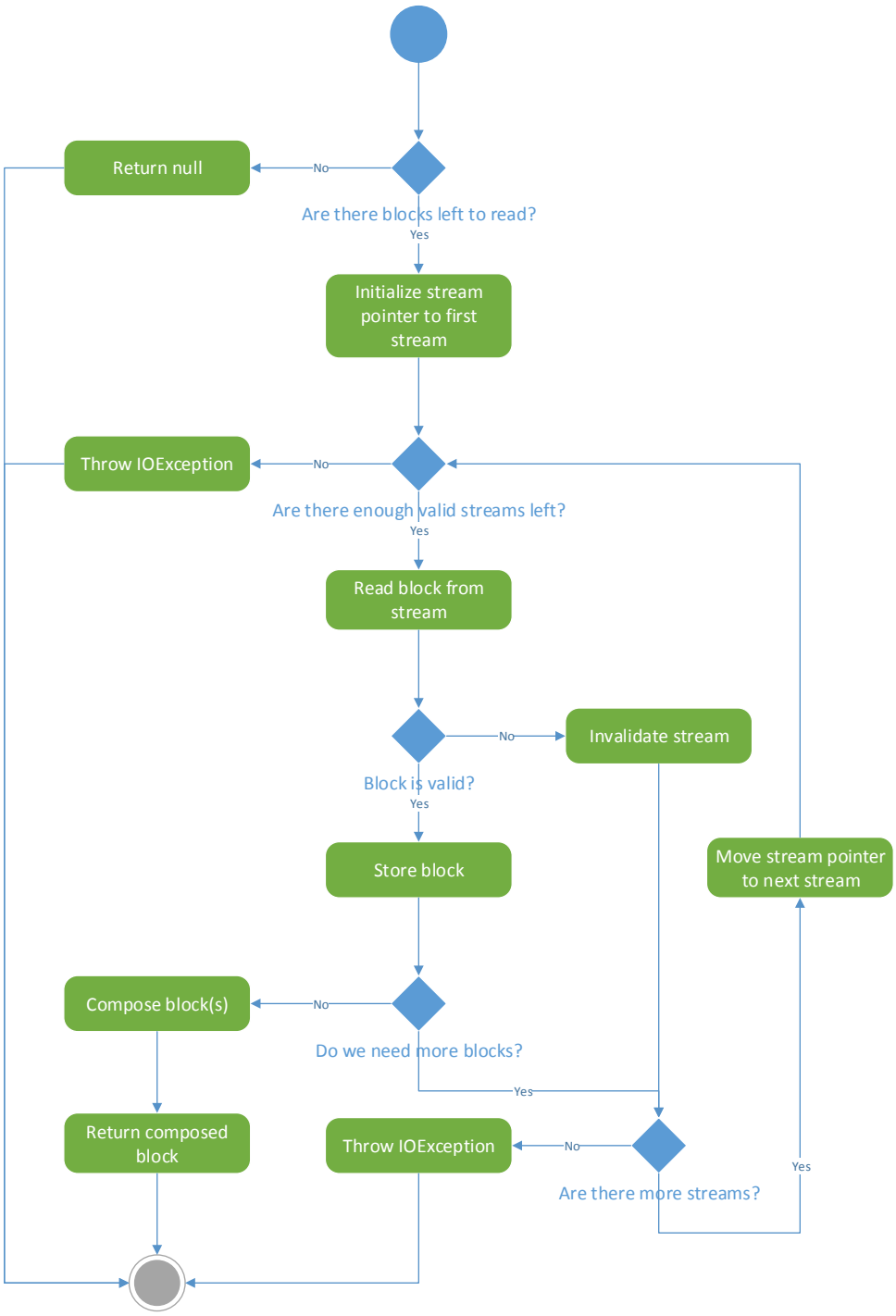


Figure 15: Activity diagram for the generateNewBlock function of the StreamComposer

execution of the function the amount of available valid streams drops below $f + 1$ then an `IOException` is thrown. This is because if there are less than $f + 1$ streams, we can no longer compose the block successfully. Finally, if we assume that out of n streams $2f$ have failed already and that we are in the last iteration of the loop (thus we already have read f blocks successfully). If the last block is not valid then it is not possible to move to pointer to the next stream (as there is none), but the check for 'more blocks required' will evaluate to true. This again results in an `IOException` being thrown.

This covers all possible flows through the function, and we have shown that it will terminate.

Verification To verify if this function works correctly, we can not simply look at the functions we have defined in Chapter 4. This is because we already implemented a reusable part of the Streaming DepSky algorithms. It is clear that the `ICompositionStrategy` implementations provide the cb function:

$$cb(x_0'^i, x_1'^i, x_2'^i, \ldots, x_f^i) = x^i$$

We also see that the `StreamComposer` acts as the c function defined in Chapter 4. Therefore, if cb is correct then c will also be correct.

However, for these fundamental functions the act of checksumming the blocks that have been read from the stream is not defined. Therefore, this component is also already a partial implementation of the Streaming DepSky algorithms. However, in the algorithms the act of composing the stream is done in a parallel fashion. To avoid reading data that we do not require, we use a sequential retrieval method. This however has some theoretical implications, but by adding some synchronicity we can avoid most, if not all, severe side effects.

5.1.6 Metadata

To store the metadata files generated by the algorithms, we require a serialization method to convert these in-memory data structures to binary files that can be stored on a file system. As a serialization method we will use Google Protocol Buffers[28]. This is a very fast, binary, serialization method which also allows messages to be extended, while maintaining backwards compatibility. Alternatives include among others: JavaScript Object Notation (JSON), Thrift and XML.

We do not consider text based serialization methods, such as JSON, to be a good fit for our purpose. Since we are storing a lot of binary data, such as the signature

[28] `https://developers.google.com/protocol-buffers/`

and block hashes, encoding these values to a text safe format increases their size considerably. Of course, compression can be used to reduce this, but that is also the case for a binary serialization method.

The Google Protocol Buffers uses messages that are defined in a text file from which the Java code is generated by the Protocol Buffers compiler. As an example we included the actual message for the Streaming DepSky-A metadata in Listing 3. The metadata has all the properties set to `required`, which indicates that the serialization will throw an exception if these fields are not set. The `BlockHash` messages included in the metadata are a `repeated` field, and it should be noted that the actual cardinality (i) of `repeated` is $i \geq 0$.

Listing 3: Protocol Buffers messages for Streaming DepSky-A metadata

```
1  message ProtoBufHighlyAvailableMetadata {
2      required int32 cloudId = 1;
3      required int32 blockSize = 2;
4      required string hashAlgorithm = 3;
5
6      repeated BlockHash blockHashes = 4;
7
8      required bytes signature = 100;
9  }
10
11 message BlockHash
12 {
13     required int32 blockNumber = 1;
14     required bytes hash = 2;
15     required int32 cloudId = 3;
16 }
```

In Figure 16 (page 51) the classes and interface that implement the storing and serialization of the metadata are shown. The highest level is the `IMetadata` interface which holds the functions that are required by other components handling metadata. Since the metadata of both algorithms overlap quite a bit, `BaseMetadata` is used as a base class for both. The serialization code is implemented by the inheriting classes since the actual serialization is a two step method. First, the actual data stored in the metadata classes has to be transferred into a Protocol Buffers message, which then can be serialized into a byte array. Thus, unfortunately we could not simply pass a object graph to the serialization method, otherwise the serialization could be abstracted using the Strategy pattern. Listing 4 shows the implementation of the `read` and `write` methods including signature verification and generation.

Listing 4: BaseMetadata read and write methods with serialization placeholders

```
1  public void write(final OutputStream out) throws IOException {
2     eraseSignature();
3
```

```
    metadataSignature = sign( serialize());

    out. write( serialize());
    out. flush();
}

public void read(final InputStream in) throws IOException
{
    deserialize( in);
    assertSignatureValid();
}

protected void eraseSignature() {
    metadataSignature = null;
}

protected void assertSignatureValid() throws IOException {
   byte[] signature = metadataSignature;

   this. eraseSignature();

   if(! strategy. verify( serialize(), signature)) {
      throw new IOException("Signature failure.");
   }
}

protected abstract byte[] serialize();

protected abstract void deserialize(final InputStream in) throws
    IOException;
```

The last interface and class represent the naming strategy of metadata files. Since it may vary per application and company how the metadata should be organized, we implemented the Strategy pattern to provide the naming of metadata files. We provide a single implementation of the `IMetadataFileNameStrategy` which simply appends ".metadata" to the filename, the `DotMetadataFileNameStrategy`. Since this class does not hold any state, it is implemented as a Singleton[29].

5.1.7 Minor Components

Quorum Executor Some commands are executed in parallel over all cloud providers and require a quorum of responses before returning. These are indicated in the pseudocode as a 'parallel for loop' with a 'wait until' condition. This logic has been placed in a single class, without any abstractions: `QuorumExecutor` (Figure 17 (page 52)). Task that are submitted to the `QuorumExecutor` should return a boolean value indi-

[29] More information about the Singleton pattern can be found in Appendix A4.4.

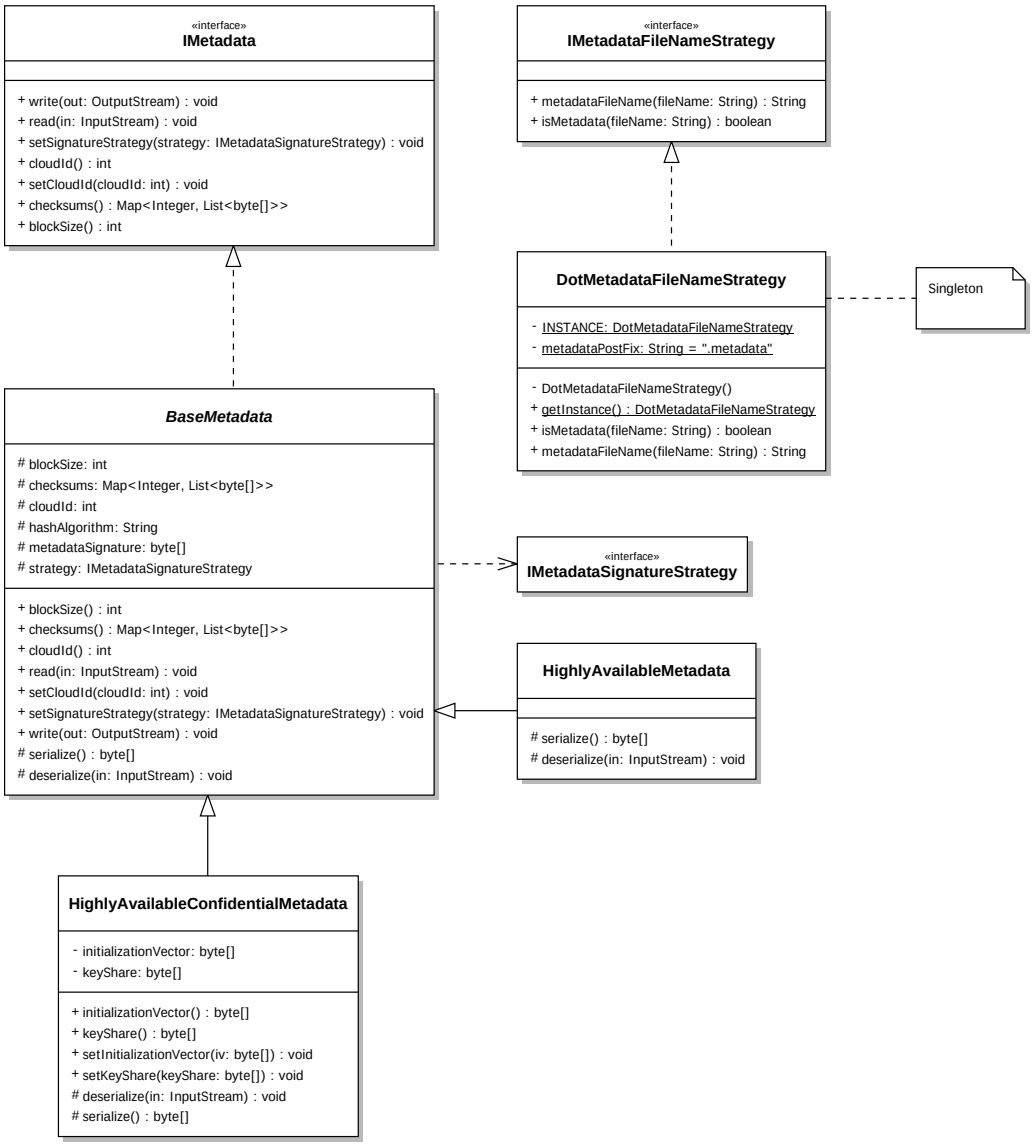


Figure 16: Class layout for Metadata representation and serialization

cating a successful executing or a failure. Any exception thrown from the task will indicate that the task has failed. Once the `QuorumExecutor` has gathered enough responses, it will return a boolean value indicating success or failure.

Essentially, the `QuorumExecutor` should never return a failure. Only if the assumed value of f in the selected set of cloud providers is invalid, e.g. the actual number of faulty cloud providers is greater than f, then a failure is returned.

Cloud Provider Abstraction To avoid having to write wrappers for the APIs of various cloud providers, we use a cloud abstraction library: JClouds[30]. However,

[30] `http://jclouds.incubator.apache.org/`

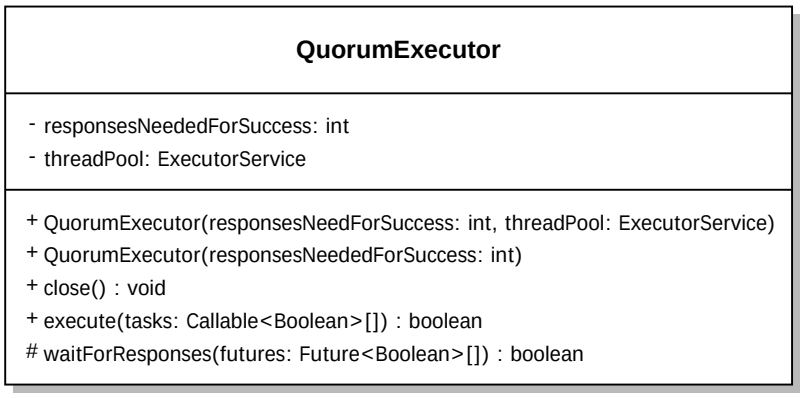


Figure 17: QuorumExecutor class diagram

due to a limitation in JClouds we also have to indicate the size of the `InputStream` that has to be saved. This is even more unfortunate if we consider that some operations, such as stream encryption and erasure coding can potentially modify the length of the stream. The attentive reader should have already noticed the `calculateOutputStreamLength` function on the `IStreamDecompositionStrategy` interface. This function is used to adjust the size of the input reported to JClouds.

To avoid consumer being directly tied to the JClouds dependency, we distribute the functionality of the `JCloudsConnection` class over multiple interfaces. These interfaces are specialized to a single operation: read, write, delete, and listing of files. This class layout is shown in Figure 18 (page 53).

5.2 Streaming DepSky-A

As we have seen in Chapter 4, the Streaming DepSky-A algorithm is also the base of Streaming DepSky-CA. Thus we have used base classes were appropriate to simplify the implementation of both the algorithms. Obviously, since these classes will tie all the components discussed earlier together, they have a lot of dependencies. We will cover the write and read algorithms in separate sections, and we see that most of the complexity is captured in the common components we implemented and discussed earlier. This allows us to keep a clear view of the actual implementation of the algorithm, and avoids us getting lost in the implementation details.

5.2.1 Write algorithm

Figure 19 (page 54) shows the classes that implement the Streaming DepSky-A write algorithm. The class exposed to the user is `HighlyAvailableFileWriter`. The actual `write` method is implemented in the base class, as it is not significantly different

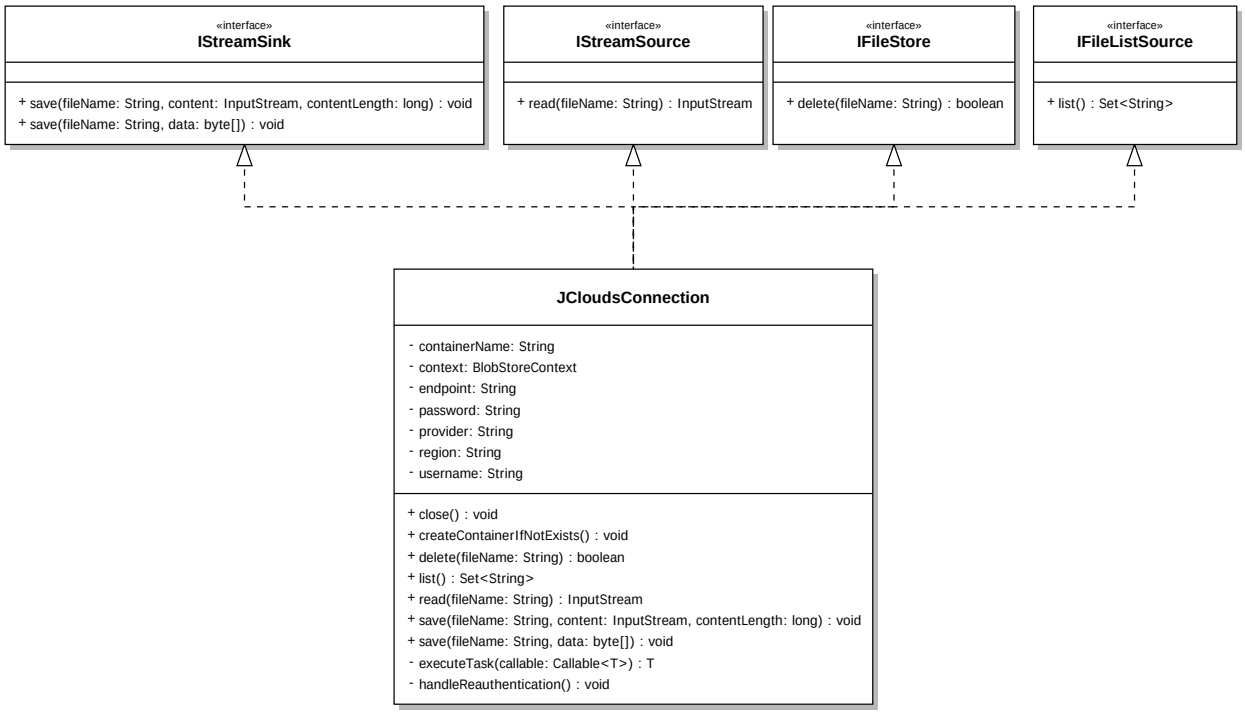


Figure 18: Class layout of the JClouds library and its abstractions

from the Streaming DepSky-CA write method. The uploading of the data is shown in Listing 5.

Listing 5: Implementation of the file saving part of the write function

```
IBlockReader reader = blockReaderFactory.construct(in);

IBlockChecksumCalculator checksumCalculator =
    blockChecksumCalculatorFactory.construct();
IStreamDecompositionStrategy decompositionStrategy =
    getDecompositionStrategy();
IStreamDecomposer decomposer = new StreamDecomposer(reader,
    checksumCalculator, decompositionStrategy);

InputStream[] inputStreams = decomposer.decompose();

// Create tasks for uploading the files
StreamWriterTask[] writerTasks = new StreamWriterTask[
    inputStreams.length];
for (int i = 0; i < inputStreams.length; ++i) {
   writerTasks[i] = new StreamWriterTask(streamSinks[i],
      fileName, inputStreams[i], decompositionStrategy.
      calculateOutputStreamLength(size, blockSize));
}
boolean fileSaved = quorumExecutor.execute(writerTasks);
```

We see a lot of object compositioning to achieve the behaviour we want. The most important logic is actually in the cooperation between the `StreamDecomposer` and

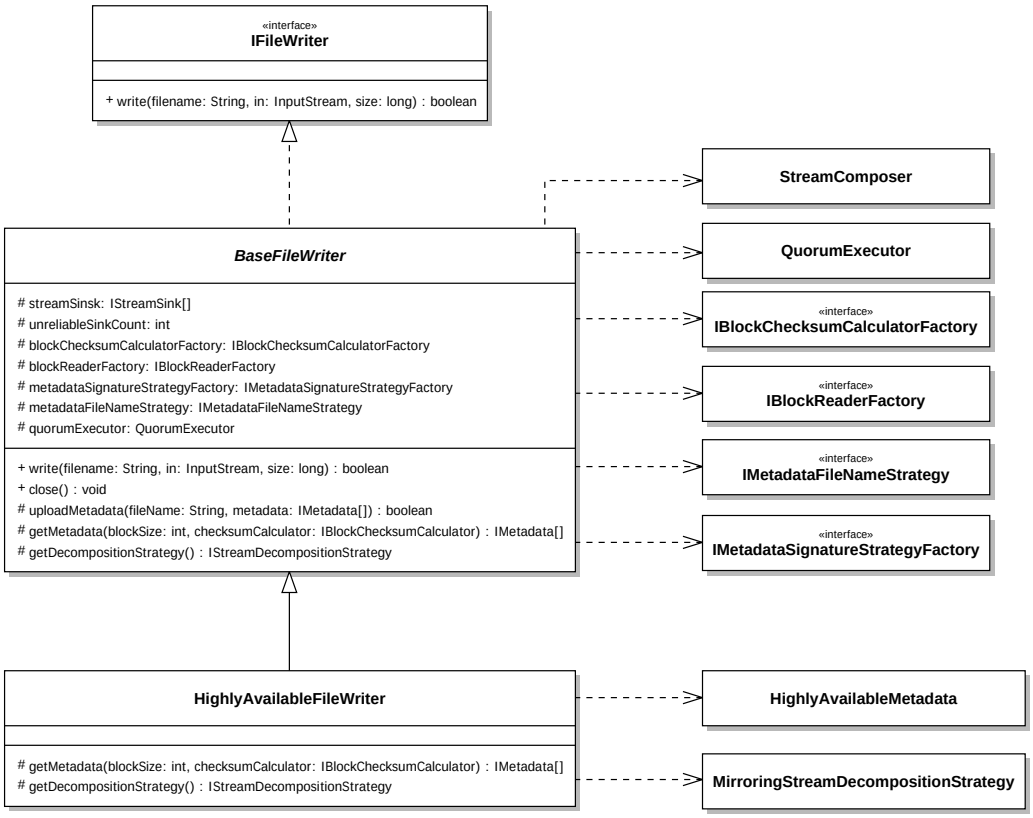


Figure 19: Class layout for the Streaming DepSky-A write algorithm

the implemented strategy. As we know, the behavior of the `StreamDecomposer` class is dependent on the provided strategy. The used strategy is returned by the `getDecompositionStrategy()` function which is implemented and configured in the `HighlyAvailableFileWriter` class. This function is shown in Listing 6.

Listing 6: Implementation of the getDecompositionStrategy function

```
protected IStreamDecompositionStrategy getDecompositionStrategy() {
   IStreamDecompositionStrategy decompositionStrategy = new
      MirroringStreamDecompositionStrategy();
   decompositionStrategy. setTotalStreams( streamSinks. length);
   decompositionStrategy. setToleratedFailures( unreliableSinkCount);

  return decompositionStrategy;
}
```

The implementation of the `decomposeBlock` function for mirroring uses a trick implemented by the `ByteBuffer` class. It is possible to copy a `ByteBuffer` without having to copy the data held in it with the `duplicate()` function. Obviously, if the data in one `ByteBuffer` changes it will change in all of them. The implementation of the `decomposeBlock` function is shown in Listing 7.

Listing 7: Implementation of decomposeBlock function in the MirroringStreamDecompositionStrategy class

```
public ByteBuffer[] decomposeBlock( ByteBuffer data) {
  if ( data == null)
    throw new ArgumentNullException("data");

   ByteBuffer[] result = new ByteBuffer[ totalStreams()];

  for (int i = 0; i < totalStreams(); ++ i) {
     result[ i] = data. duplicate();
  }

  return result;
}
```

The saving of the metadata is a very simple operation that consists of extracting the information from the `IBlockChecksumCalculator`, and including information such as the block size and hash algorithm. These metadata files are then serialized and uploaded using the `QuorumExecutor`. The creation of the metadata is done in the `HighlyAvailableFileWriter` class, as this is specific to the implementation.

Verification First, we note that the values of n and f are configured when the `HighlyAvailableFileWriter` is configured. n is derived from the amount of supplied `IStreamSink` instances, and f is specified by the user. We assume that the condition $n \geq 3f + 1$ holds in all cases.

The implementation and the pseudocode are not significantly different. The first difference between the implementation and the pseudocode is found on line 14 of Algorithm 1. We do not have an abstraction for the clouds that accepts blocks of data for a given file. The abstraction we use reads from an `InputStream` instance and writes that to the cloud. Internally, the block is read from the underlying stream, mirrored and then made available through each `InputStream`. Since the write to a single cloud can only fail once, the semantics of the implementation match that of the algorithm.

We conclude that if the parameters n and f are configured correctly, that Theorem 1 is valid for our implementation.

5.2.2 Read algorithm

As the write algorithm, the read algorithms of both Streaming DepSky-A and Streaming DepSky-CA have a lot in common. Therefore we will again use a base class, shown in Figure 20 (page 56), which implements the Streaming DepSky-A algo-

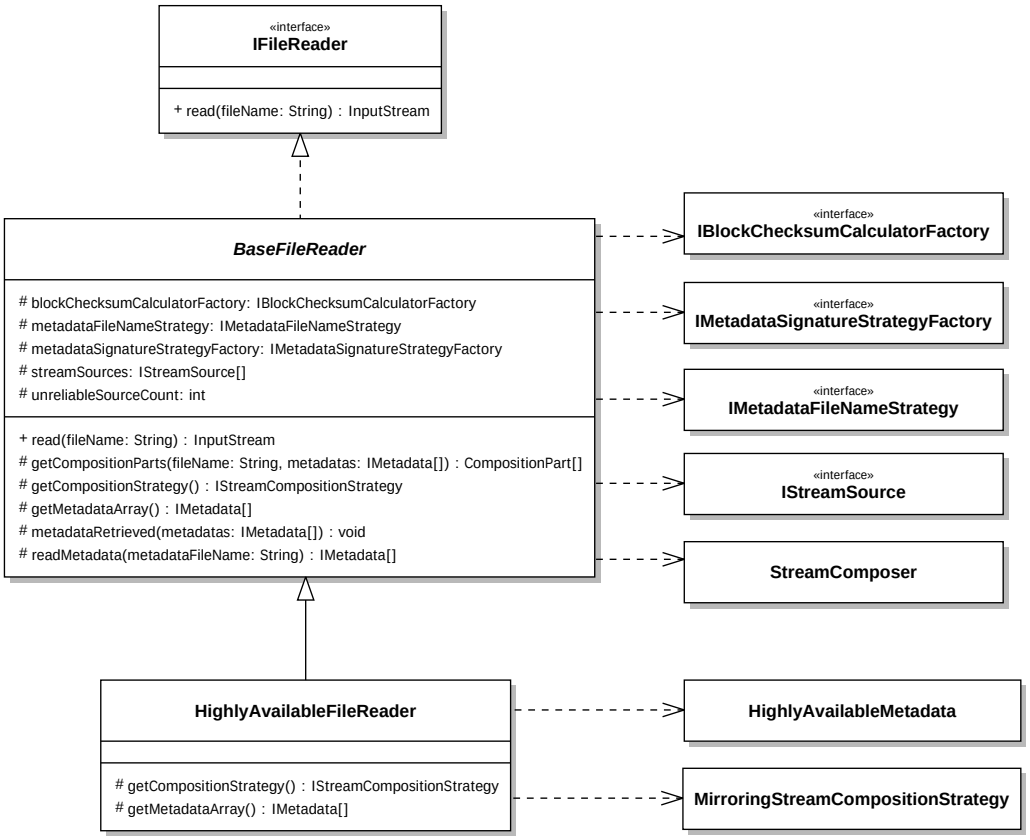


Figure 20: Class layout for the Streaming DepSky-A read algorithm

rithm, with only a few specific parts regarding metadata and composition being handled by the actual exposed implementation in the `HighlyAvailableFileReader` class.

The actual implementation of the read algorithm is shown in Listing 8. First, an attempt is made to retrieve the metadata associated with the given filename. If this fails, an exception is thrown. If it succeeds, the set of cryptographic hashes is retrieved from the metadata and the stream associated with the file stored in each cloud is wrapped in an `IBlockReader`. The hashes and the reader are then exposed as a `CompositionPart`. These `CompositionParts` are then passed to the `StreamComposer` along with the `IStreamCompositionStrategy` and an instance of a `IBlockChecksumCalculator` to allow the `StreamComposer` to verify the checksums of the blocks. Finally, the resulting `InputStream` is retrieved and returned by a call to the `compose()` function.

Listing 8: Retrieval and verification of metadata

```
1 public InputStream read( String fileName) throws IOException {
2    String metadataFileName = metadataFileNameStrategy.
        metadataFileName( fileName);
3
4   // Read all metadata files from all clouds
```

```
5     IMetadata[]  metadatas =  readMetadata( metadataFileName);
6
7    // Did we retrieve any metadata?
8    if ( Arrays. equals( metadatas, new  IMetadata[ metadatas. length]))
9    {
10     throw new  IOException(); //File does not exist
11   }
12
13    CompositionPart[]  compositionParts =  getCompositionParts(
        fileName,  metadatas);
14    IStreamCompositionStrategy  compositionStrategy =
        getCompositionStrategy();
15    IStreamComposer  composer = new  StreamComposer(
        compositionStrategy,  compositionParts,
        blockChecksumCalculatorFactory. construct());
16
17   return  composer. compose();
18 }
```

The `IStreamCompositionStrategy` implementation used is very simple. Since we are mirroring each block, we can simply return the first valid block as the result block. This is shown in Listing 9.

Listing 9: Implementation of the composeBlock function

```
1 public  ByteBuffer  composeBlock( ByteBuffer[]  data) {
2   for(int  i = 0;  i <  data. length; ++ i) {
3     if( data[ i] != null) {
4       return  data[ i]. duplicate();
5     }
6   }
7
8   return null;
9 }
```

Since the integrity of each block submitted to the `composeBlock` function has already been verified, we can see that this implementation of the cb function is valid.

Verification We see that the implementation of the read algorithm is actually not a lot of code. Most complexity resides in the `StreamComposer` which we already verified to be working correctly with the assumption that the cb function was valid. As we have seen, this is the case. Thus we can very quickly conclude that the implementation of the Streaming DepSky-A read algorithm is correct.

5.3 Streaming DepSky-CA

As we saw in Chapter 4, we require some additional components to implement the Streaming DepSky-CA algorithm successfully. These are the erasure encoding and

decoding routines, the encryption and decryption of the streams, and the secret secret sharing scheme.

Initially, we wanted to keep the library as pure as possible. Meaning, that we did not want to include (dependencies on) native code in the library. Unfortunately, we were unable to gain satisfactory performance when using a erasure coding scheme implemented in Java. Therefore, we had to look at a library written in C, JErasure[31]. The library and its functions are documented very well [29], and even if a complete rewrite is pending[32] using the improved routines introduced in [30], we feel confident enough to use it.

Unfortunately, it is not possible to use native code directly in Java. It requires a wrapper implementation in native code to convert from and to Java primitives. Since there are no wrapper implementations available, we have to implement these ourselves. To keep things simple, we implemented only a very thin wrapper that converts parameters and return values from Java to C and vice versa. Thus the provided Java API is very rough, and is essentially a one-to-one mapping of the C functions. Finally, we also include some minor changes that allow the JErasure library and wrappers to be compiled easily on the Windows operating system. The source code is published and available on GitHub[33]. Even with the translation between native code and Java, the performance is orders of magnitude better than the pure Java implementation.

In Figure 21 (page 59) the general class layout for the erasure coding and decoding is shown. The actual erasure coding algorithm used is the Reed-Solomon error correction code. The `ReedSolomon` and `Jerasure` classes provide the coupling to the native code and are part of the Java wrapper for the JErasure library. Internally we split the block that has been read into $f + 1$ pieces, thus the actual block size should be a multiple of $f + 1$ to allow the block to be split. Since `IStreamDecompositionStrategy` implementations can be queried for a adjusted block size depended on the actual chosen block size. This can be done by invoking the `getRecommendedBlockSize` function, it is defined as $(f + 1) \times \beta$. If we assume that $f = 1$ and $\beta = 8192$ then we see that if we want to read a file of size 17721, that we end up with one full read of $8192 \times 2 = 16384$ which can be encoded since $16834 \bmod 2 = 0$ and one block of size 1337 which is clearly not a multiple of 2. To avoid this we require a padding scheme, which has the unfortunate consequence in that this increases the length of the output stream.

Our padding scheme is much like the padding schemes introduced in [31] and [32]. Except that we operate on much larger block sizes than 16 or 32 bytes, and thus adding a complete block of padding (which in our example would be 16 KiB) is quite

[31] `http://web.eecs.utk.edu/~plank/plank/papers/CS-08-627.html`
[32] Based on personal correspondence
[33] `https://github.com/jvandertil/Jerasure`

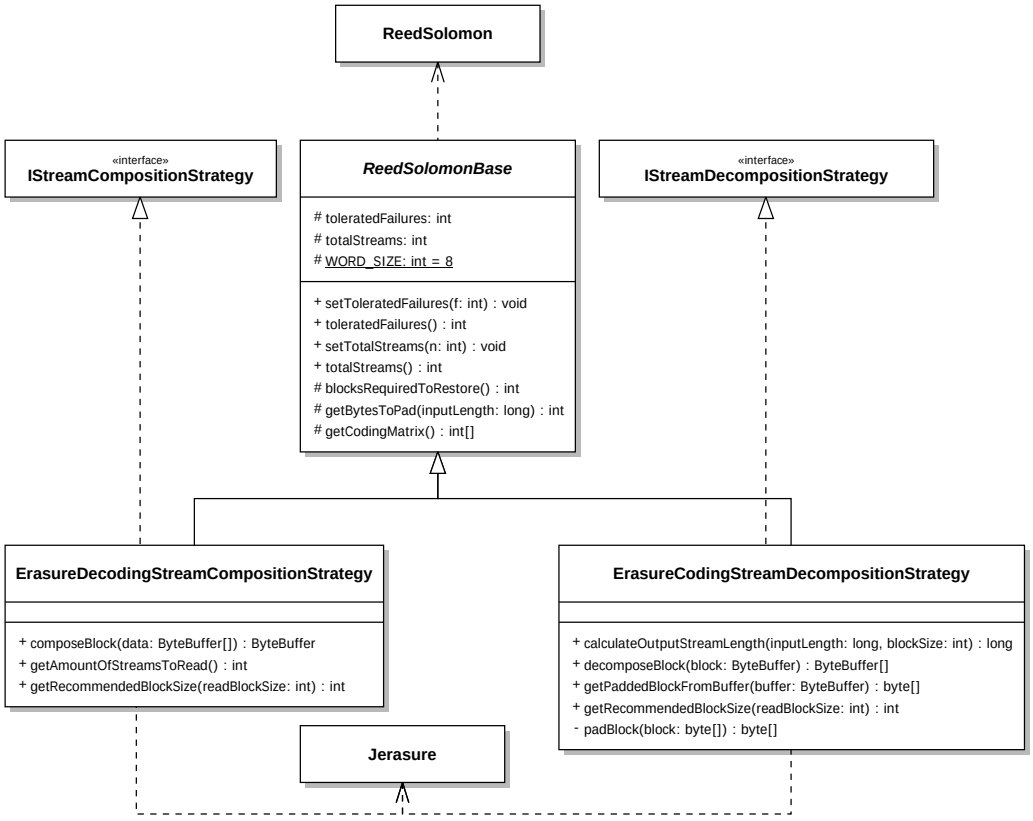


Figure 21: Class layout for the erasure coding and decoding functionality

a significant waste of space. Thus, we pad each block that has been read. However, we also have to know this when decoding the blocks. Our padding scheme is as follows: We have a stream of length l_s and a block size of β. The number of bytes to pad per block is:

$$l_P = F_p(\beta) = (f+1) - (\beta \bmod (f+1))$$

The value of the bytes padded is equal to l_P, which is sufficient until f reaches 126 since Java uses signed bytes. We will use the adjusted block size:

$$\beta_R = \beta \times (f+1)$$

since this will be the size of the blocks read from the stream. The total length of each output stream (there are $n = 3f+1$ output streams), assuming $N_\beta = \frac{l_s}{\beta_R}$ is:

$$\sum_{i<N_\beta} \frac{\beta_R + F_p(\beta_R)}{f+1}$$

In our implementation we however used an optimization of the calculation, which does not require the summation as a loop (see Listing 10). This is time-memory trade-off.

Listing 10: Calculation of output stream length

```
public long calculateOutputStreamLength(long inputStreamLength, int blockSize) {
  long totalOutputSize = 0;
  long readBlockLength = getRecommendedBlockSize( blockSize);

  long sizeOfLastBlock = ( inputStreamLength % readBlockLength);
  long numberOfBlocksRead = (( inputStreamLength - sizeOfLastBlock) / readBlockLength);

  long outputBlockSize = ( readBlockLength + getBytesToPad( readBlockLength)) / blocksRequiredToRestore();
   totalOutputSize += ( numberOfBlocksRead * outputBlockSize);

  if ( sizeOfLastBlock > 0) {
     totalOutputSize += ( sizeOfLastBlock + getBytesToPad( sizeOfLastBlock)) / blocksRequiredToRestore();
  }

  return totalOutputSize;
}
```

The encryption of the streams is done by utilizing the classes offered by the JDK. The encryption ciphers are implemented by subclasses of the `Cipher` class. And the specialized `CipherInputStream` will transparently filter blocks of bytes through the encryption cipher before returning them to the caller.

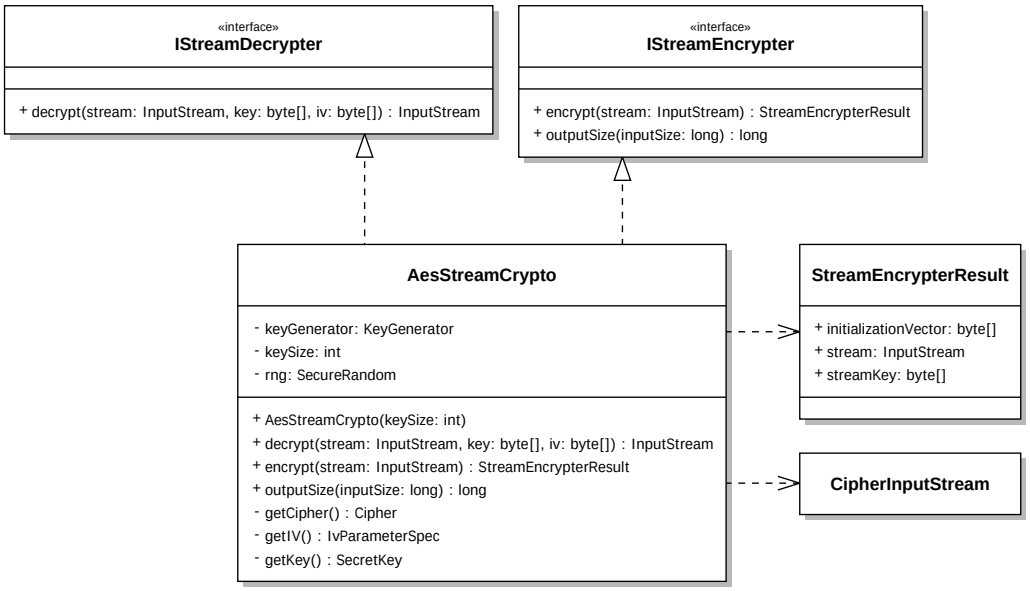


Figure 22: Class layout for the encryption and decryption of streams

In Figure 22 (page 61) the class layout is shown. The encryption cipher used is the Advanced Encryption Standard cipher (also known as Rijndael). The classes are compatible with both 128 and 256 bit keys, assuming that the underlying Java Runtime Environment allows this[34]. The AES block mode is configured as Cipher Block Chaining (CBC). This requires an **I**nitialization **V**ector (IV) as each block i is XOR'ed with block $i - 1$, where the IV will take the place of block -1. The encrypted stream, the generated key, and the initialization vector are encapsulated in a `StreamEncrypterResult` class and returned to the caller.

The Secret Sharing Scheme is implemented by the open source library SecretShare-Java[35], specifically the `SecretShare` class. To make the API easier to work with, we created a small wrapper around it that exposes three simple, self explanatory, functions as shown in Figure 23 (page 62). Internally, the library will convert a byte array into a `BigInteger` object. Since the library only supports positive `BigInteger` objects, so a function is provided that can determine if a given byte array can be encoded by the `encodeKey` function.

The functions exposed by the `SecretSharingHelper` closely match the fundamental functions we defined in Chapter 4. We see that `encodeKey` implements:

$$sss(D, n, k) = [D_0, D_1, D_2, \ldots, D_n]$$

[34]For 256 bit keys the Unlimited Strength Jurisdiction files are required. 128 bit keys should always be available

[35]`http://sourceforge.net/projects/secretsharejava/`

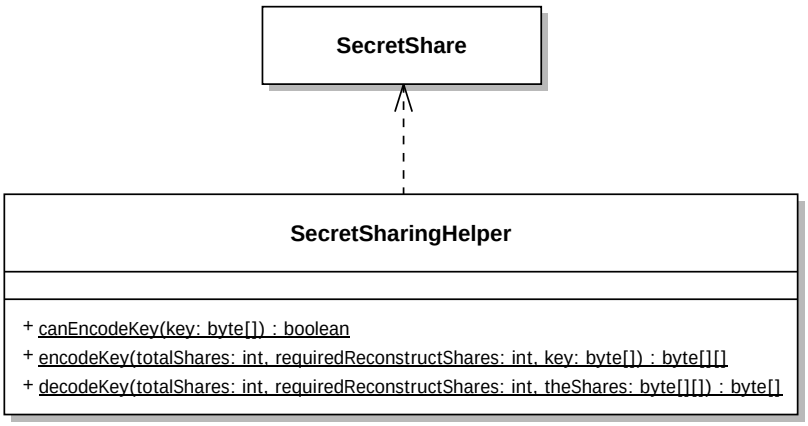


Figure 23: Class layout for the Secret Sharing Scheme

and that `decodeKey` implements:

$$sss^{-1}(D_0, D_1, D_2, \ldots, D_i) = D$$

However, we see that the `decodeKey` function requires a little more information, namely the parameters given to the `encodeKey` function, than the fundamental function we defined.

5.3.1 Write algorithm

In Figure 24 (page 66) the class layout of the implementation of Streaming DepSky-CA write algorithm. Since the algorithm has to store a small amount of state between the encryption of the stream and the uploading of the metadata, two `ThreadLocal` variables are introduced. Both the secret key and the IV are split using the Secret Sharing Scheme. While it is not necessary to keep the IV as secret as the key, keeping as much information secret from a (would be) attacker as possible is common sense.

Listing 11: Implementation of the write method

```
1 public boolean write(final String fileName, final InputStream
      in, final long size) throws IOException {
2   try {
3     StreamEncrypterResult encrypted = streamEncrypter.encrypt(in);
4     keyShares.set(SecretSharingHelper.encodeKey(streamSinks.
        length, unreliableSinkCount + 1, padForSecretSharing(
        encrypted.streamKey)));
5     ivShares.set(SecretSharingHelper.encodeKey(streamSinks.
        length, unreliableSinkCount + 1, padForSecretSharing(
        encrypted.initializationVector)));
6
7     boolean result = super.write(fileName, encrypted.stream,
        streamEncrypter.outputSize(size));
```

```
8
9       return  result;
10    } finally {
11       keyShares. remove();
12       ivShares. remove();
13    }
14 }
```

Listing 11 shows the implementation of the `write` method. As we noted earlier the Secret Sharing Library does not support byte arrays that are converted in a negative `BigInteger` object. Thus, we use a very simple padding scheme to assert that the key always results in a positive `BigInteger`.

This does result in a leak of information, as an attacker can now know the first byte of the padded key. However, while I do not have any evidence to support this, I feel that this is a better approach than not using keys that result in a negative `BigInteger`. If we do not allow keys that result in negative `BigIntegers` than we decrease the search space by half. Thus the key search area would no longer be 2^{128} or 2^{256} but 2^{64} and 2^{128}.

Verification The verification of the Streaming DepSky-CA algorithm is very limited. Due to our implementation the bulk of the verification is already done, as it reuses all the components from Streaming DepSky-A. We see that the stream is encrypted using the StreamEncrypter, and that from there on the process is exactly the same as Streaming DepSky-A. The `getMetadata` function has access to the `ThreadLocal` storage for both the IV and the secret key.

5.3.2 Read algorithm

Figure 25 (page 67) shows the class layout for the Streaming DepSky-CA read algorithm. And Listing 12 shows the actual implementation of the read method. It is clear that this is a very simple function which leans heavily on the Streaming DepSky-A read algorithm.

Listing 12: Implementation of the Streaming DepSky-CA read function

```
1 public  InputStream  read( String  fileName) throws  IOException {
2    try {
3       InputStream  composed = super. read( fileName);
4       InputStream  decrypted =  streamDecrypter. decrypt( composed,
          key. get(),  iv. get());
5       return  decrypted;
6    } finally {
7       key. remove();
8       iv. remove();
```

```
  }
}
```

An important piece of functionality is done by the `metadataRetrieved` method. This function is called after the metadata has been retrieved from the cloud, and before the file is retrieved. The base implementation is a no-op. However, we use it to intercept the metadata and retrieve the key and IV. These are then stored in `ThreadLocal` variables, so that these are available when the decrypted stream has to be returned. The implementation is shown in Listing 13.

Listing 13: Implementation of the metadataRetrieved method

```
protected void metadataRetrieved( IMetadata[] metadatas) {
  byte[][] keyShares = new byte[ streamSources. length][];
  byte[][] ivShares = new byte[ streamSources. length][];

  for (int i = 0; i < metadatas. length; ++ i) {
     HighlyAvailableConfidentialMetadata metadata = (
        HighlyAvailableConfidentialMetadata) metadatas[ i];
    if ( metadata != null) {
       keyShares[ metadata. cloudId()] = metadata. keyShare();
       ivShares[ metadata. cloudId()] = metadata.
          initializationVector();
    }
  }

   key. set( getUnpadded( SecretSharingHelper. decodeKey(
      streamSources. length, unreliableSourceCount + 1, keyShares)));
   iv. set( getUnpadded( SecretSharingHelper. decodeKey( streamSources.
      length, unreliableSourceCount + 1, ivShares)));
}
```

Verification We see that the implementation is very simple. The only difference is that the stream is decrypted before being returned. We also see that if at least $f + 1$ metadata is returned, that the key and IV can be retrieved using the secret sharing scheme. Again, the erasure coding and composition is handled by the `IStreamCompositionStrategy` and `StreamComposer`.

5.4 Future Work

The implementation of the Streaming DepSky algorithms can be improved and extended with some additional functionality. For example, a garbage collection scheme can be considered. That way all files which are not stored properly (e.g. not on $n-f$ clouds, or without metadata) can be deleted to save storage space. One can envision this as something that runs once a month and reports all found files, after which a system administrator can decide to delete these manually or automatically.

Another improvement would be to allow file corruption to be detected and repaired while the system is running. Thus, if corruption in a file on a cloud provider is detected then as long as the files are valid on $f + 1$ cloud providers, we can restore the file. This way it is possible to counter so called 'bit rot' or 'digital decay' on the storage systems of the cloud provider. This is not a theoretical concern, as we only expect that f cloud providers out of the n chosen will corrupt files. However, having a practical solution at hand is not a bad idea.

If it is decided that certain cloud provider is not reliable enough, or is no longer trusted. Then it should be possible to migrate the data from that cloud provider to a new one. Currently, one would have to download the data, delete the data, and reupload the data to a new set of cloud providers. This is quite a hassle, and thus a automated system that performs these tasks in a more efficient manner would be a good addition.

Finally, should the system reach its end of life, then the data is not locked in to it. It can be retrieved before the system is shut down. Since the data is stored in the cloud, it is possible to access it from where ever it is required. Also, there are no modifications done to the data when it is stored. Thus, the end user controls the format in which data is saved. As long as the end user knows how to handle the format of the data he/she saved, the data is not locked in.

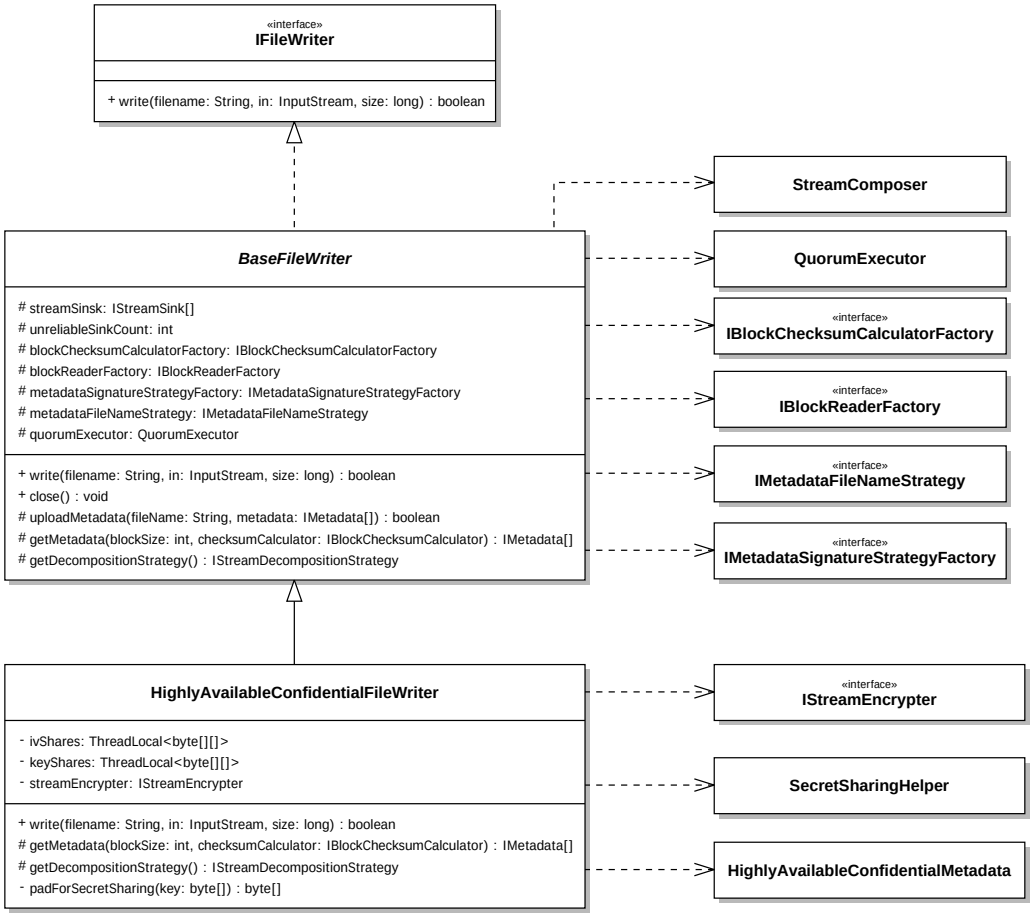


Figure 24: Class layout for the Streaming DepSky-CA write algorithm

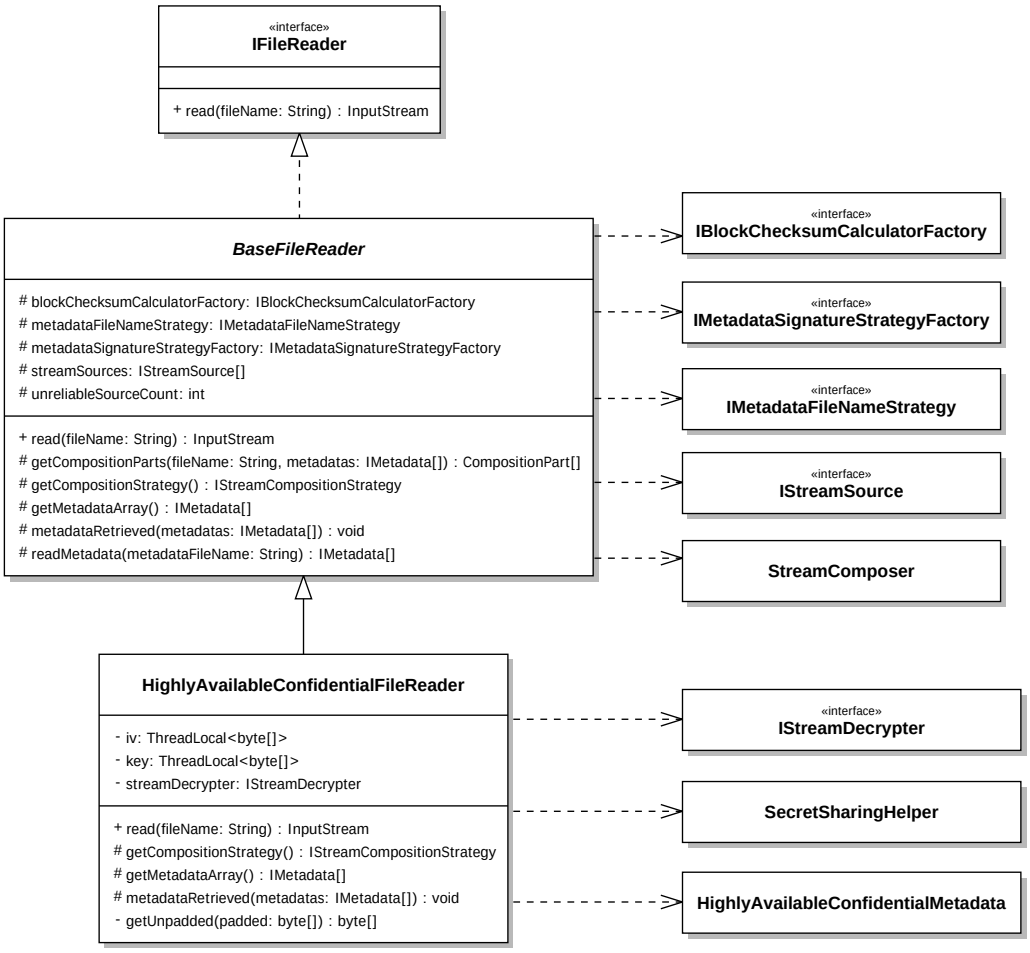


Figure 25: Class layout for the Streaming DepSky-CA read algorithm

6 Experiments

In our experiments we used the smallest amount of cloud providers possible, thus $n = 4$ and $f = 1$. For our experiments we use three established cloud storage providers, namely Amazon, Microsoft Windows Azure, and RackSpace. The fourth is a relatively new cloud provider established in Iceland, GreenQloud. The availability zones used are: eu-west-1 (Amazon), Western Europe (Azure), Dallas (TX, USA) (RackSpace). GreenQloud does not support choosing a availability zone, and thus we use the default: IS-1 (Iceland). To evaluate the performance of our implemented storage library, we envision three test scenarios.

The first will give a baseline on the performance of the individual cloud providers. Therefore, we needed a test application that runs a standardized test on only a single provider. Since the cloud abstraction used in the implementation of the library might influence the perceived performance of the clouds, we will use JClouds in the storage performance tests as well. The test environment is illustrated in Figure 26 (page 68).

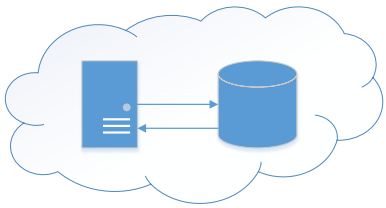


Figure 26: Test environment in the cloud.

The second set of tests will be in the setup that we consider to be the most likely deployment. The test is run on a virtual machine hosted in the test lab at TNO Groningen. From there data will be stored in all four clouds. Unfortunately, the internet connection of the server is limited to 100 MegaBits. The test environment is illustrated in Figure 27 (page 68).

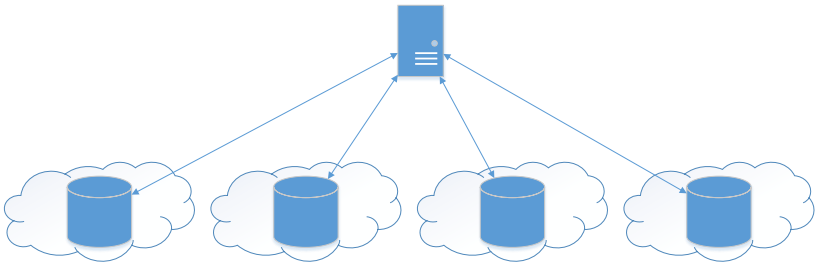


Figure 27: Cloud test environment on premise.

Finally, the third test will determine to connection speed between the cloud providers. The storage library is run on a virtual machine hosted in each cloud, and will perform the same test as the previous set. Thus, we can see the performance impact of the storage library in a bit more detail. As the limit of the cloud storage itself has been evaluated in the first set of tests. The test environment is illustrated in Figure 28 (page 69).

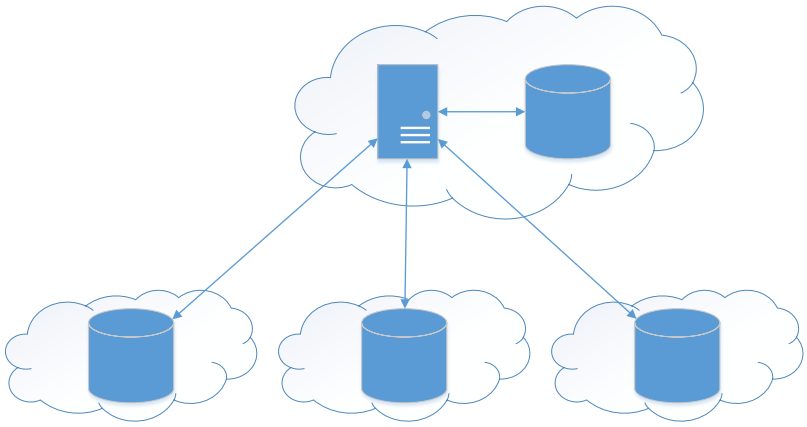


Figure 28: Cloud test environment in the cloud.

6.1 Cloud Storage Performance Tests

First we will discuss the implementation and design of the test application used. The most important classes from which the test application consists are shown in Figure 29 (page 70).

The Application class contains the entry point of the application, the `main` function. The test application is configured by passing a configuration file, file container name, and output file as command line arguments. An example configuration file for RackSpace is shown in Listing 14. Since cloud storage providers require a container (much like a top-level folder) in which files are stored by default, this has to be specified as well. The output file will contain the results of all the tests as Comma Separated Values (CSV).

Listing 14: Example test application configuration file

```
provider= cloudfiles- us
username= josvdtil
accesskey= d41d8cd98f00b204e9800998ecf8427e
```

We are interested in the performance of four specific HTTP verbs: PUT, GET, DELETE and LIST. These will upload a file, download a file, delete a file, or list a directory,

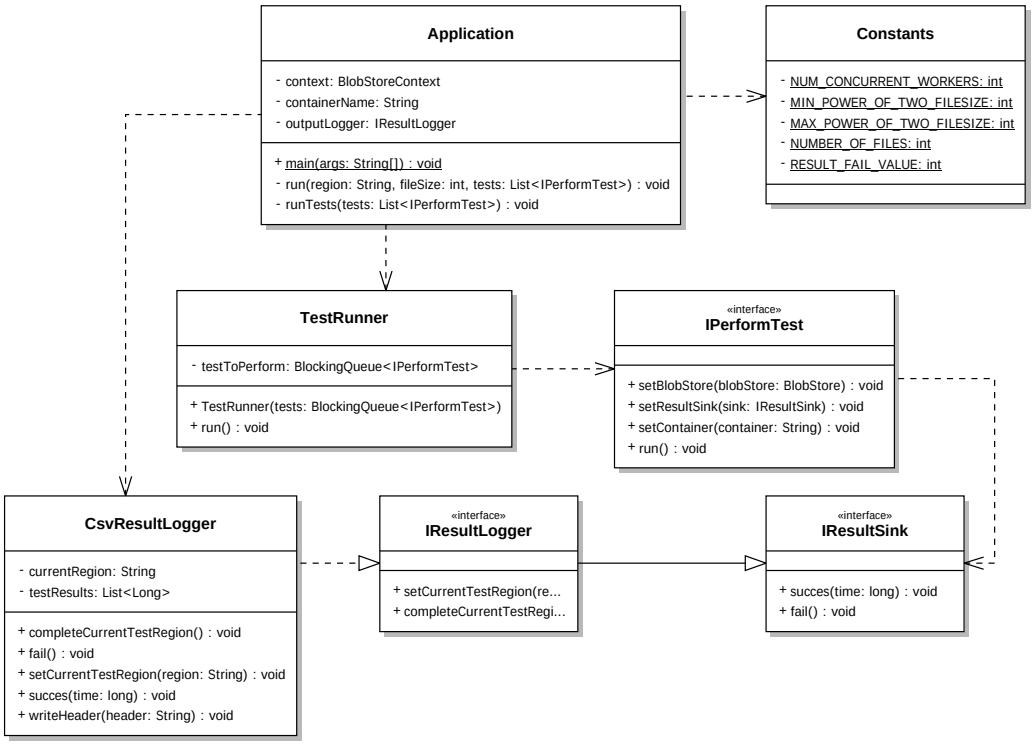


Figure 29: Test application class layout

respectively. This results in the class layout shown in Figure 30 (page 71), where each verb is represented by a single class.

The functions on the `AbstractPerformTest` class are called by the `run` method in the order shown in Figure 31 (page 72). The run method of each test is invoked by a `TestRunner`, of which we used 16 in our tests. The number of tests were 1024 for each HTTP verb. This was done for all the filesize starting at 2^0 bytes up to (including) 2^{25} bytes. The total amount of data transferred in a single direction is roughly $1024 \times 2^{26}\ \mathrm{b} = 64\ \mathrm{GiB}$.

To remove as much confounding elements as possible, these tests are run using a **V**irtual **M**achine (VM) started in the same cloud and availability zone (if applicable) as the storage. We also standardize the operating system and JDK version to the CentOS Linux Operating System 6.3 and the Oracle JDK 7 update 17.

Since not all providers offer the same specifications for their cloud servers, we are unable to standardize a single hardware configuration for all cloud providers. We can, however, standardize our tests on a single hardware specification per cloud provider. We will choose these by running the test on different virtual machine sizes, while monitoring system load and memory usage. When the bandwidth consumption (measured in Megabits or Gigabits) seems to reach a maximum, we assert that the system has enough remaining capacity. In all cases we want to assert that

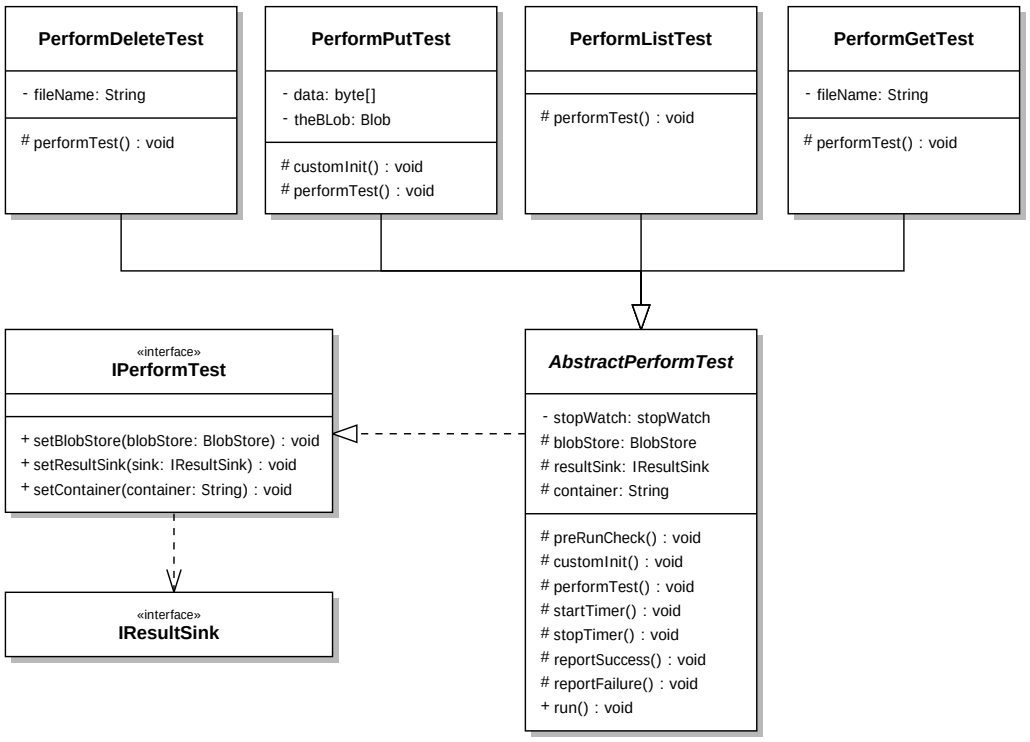


Figure 30: Classes that implement the executable tests

the CPU or the memory are not forming a bottleneck.

In most cases we found that the network is either slow, or capped at around 1 GigaBit per second. Only Amazon's provided a network connection that was not capped. Thus every time the network seemed to have reached its maximum throughput, we found that actually the system was the bottleneck. Indeed, it is possible to reach upload speeds far exceeding anything the other cloud providers have to offer. Upload speeds of over 1 GigaByte per second have been reported in the Amazon Cloud [33].

Thus, to avoid giving Amazon too big a benefit from running on super fast hardware (which was roughly 10 times as expensive as the other clouds), we chose a machine with specifications more in line with the other cloud providers. The specifications for all cloud providers are shown in Table 5.

Cloud	Instance Type	Cores	RAM
Amazon	m3.2xlarge	8	30 GB
Azure	Extra Large	8	14 GB
GreenQloud	Large	8	8 GB
RackSpace	–	6	15GB

Table 5: Virtual Machine Specifications

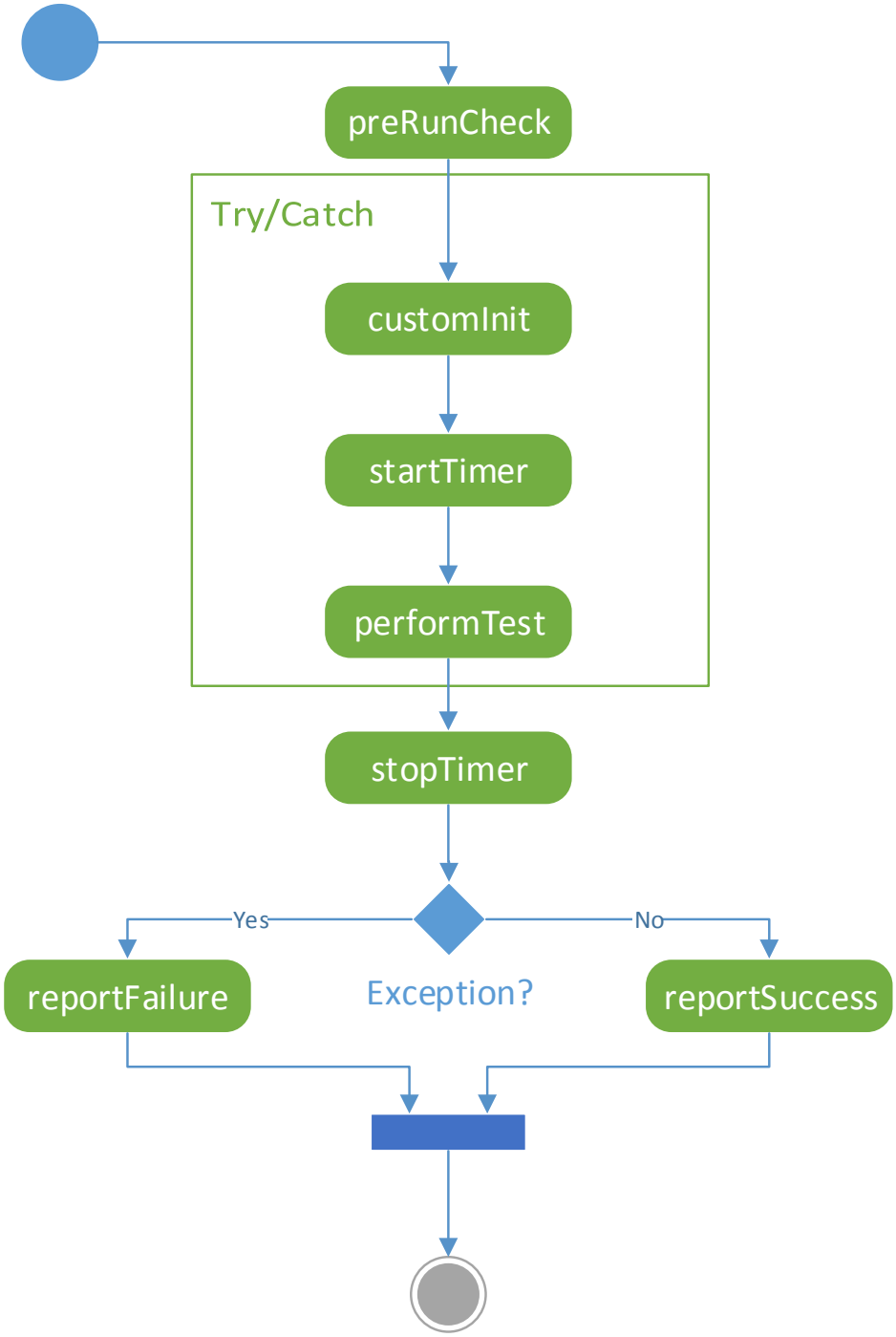


Figure 31: Activity diagram for the run method of AbstractPerformTest

6.1.1 Results

The data was gathered between April 13, 2013 and April 21, 2013. The overall availability statistics are shown in Table 6. Across the board we see that availability is not 100%, but we are very close too it. An interesting observation is that Windows Azure seems to the worst cloud storage provider of all. This is strange since in a recent industry report [34], Azure was voted the best cloud provider based on availability.

Cloud Provider	Measurements	Failures	Availability
Amazon	3 853 850	390	99,9999 %
Azure	3 427 524	1257	99,9996 %
GreenQloud	2 974 600	518	99,9998 %
RackSpace	3 914 754	428	99,9998 %

Table 6: Overall statistics per cloud provider

When we take a closer look (Figure 32 (page 74)), we see that only when we reach the filesize of 2^{25} bytes (or 32 MiB) that the success rate of the PUT verb drops dramatically. Indeed, we see that up until that file size not a single request has failed. However, we that both Amazon and RackSpace are also experiencing a bad success rate at the 32 MiB file size.

Interesting observations are that RackSpace was not able to attain a success rate for 100% for any file upload. And that GreenQloud was the only cloud provider that has failing LIST requests. All providers were also able to execute the DELETE requests with a 100% success rate.

When we compare the throughput of the providers when using the GET verb, in Figure 33 (page 75), we see significant differences between the providers. Amazon achieves the highest absolute throughput, which was expected as the Amazon network is very fast. Windows Azure comes in second in absolute throughput. Azure could have scored better, but the virtual machines were limited to slightly above 1 GigaBit per second. RackSpace comes in third, since the network was limited to around 600 MegaBits per second. We have no explanation why GreenQloud performs how it does. Either the storage backend is extremely slow, the network is highly congested, or the virtual machine network interface is severely limited.

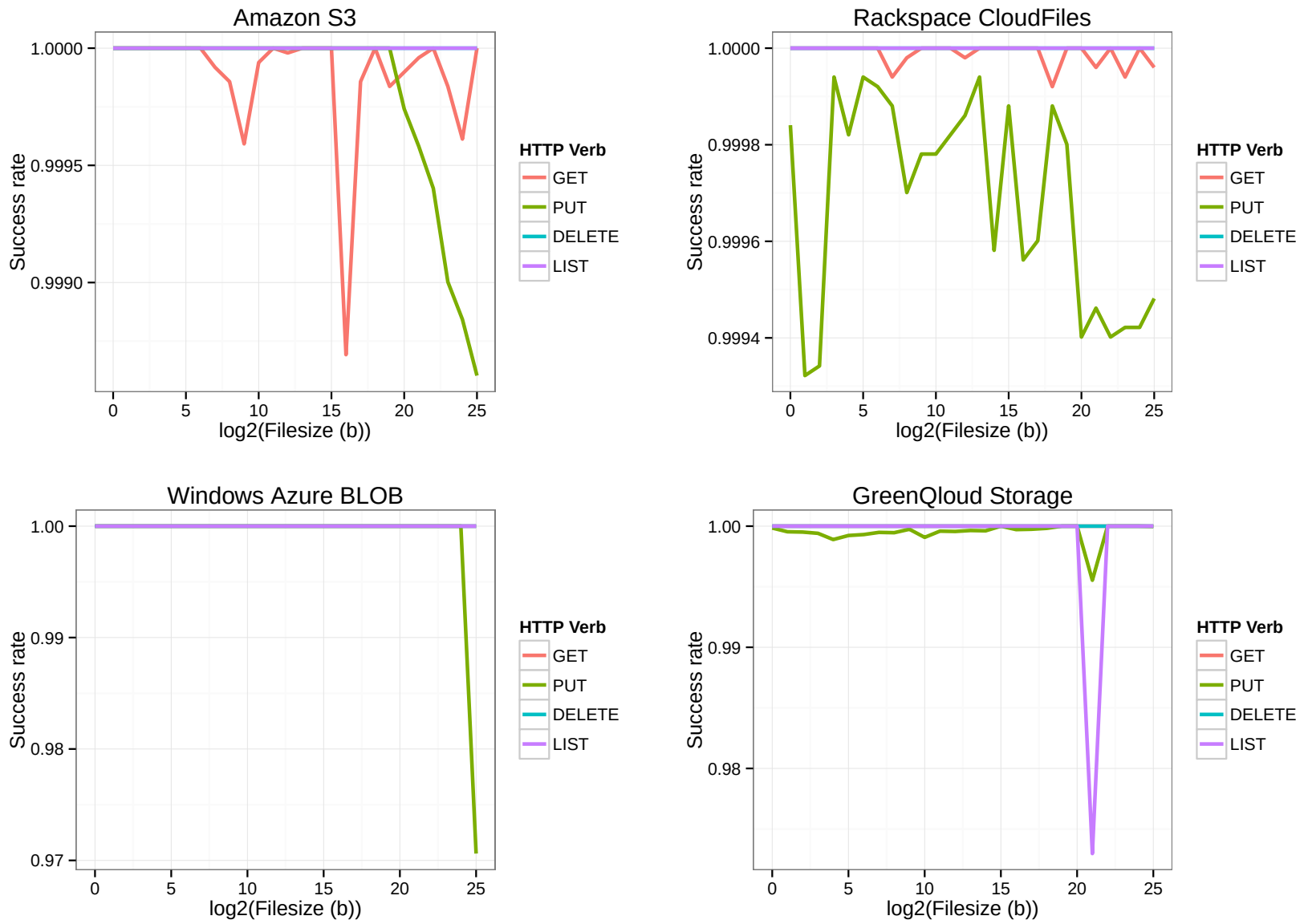


Figure 32: Availability per HTTP verb. Note that the scales are not equal, this is on purpose as the differences are very small.

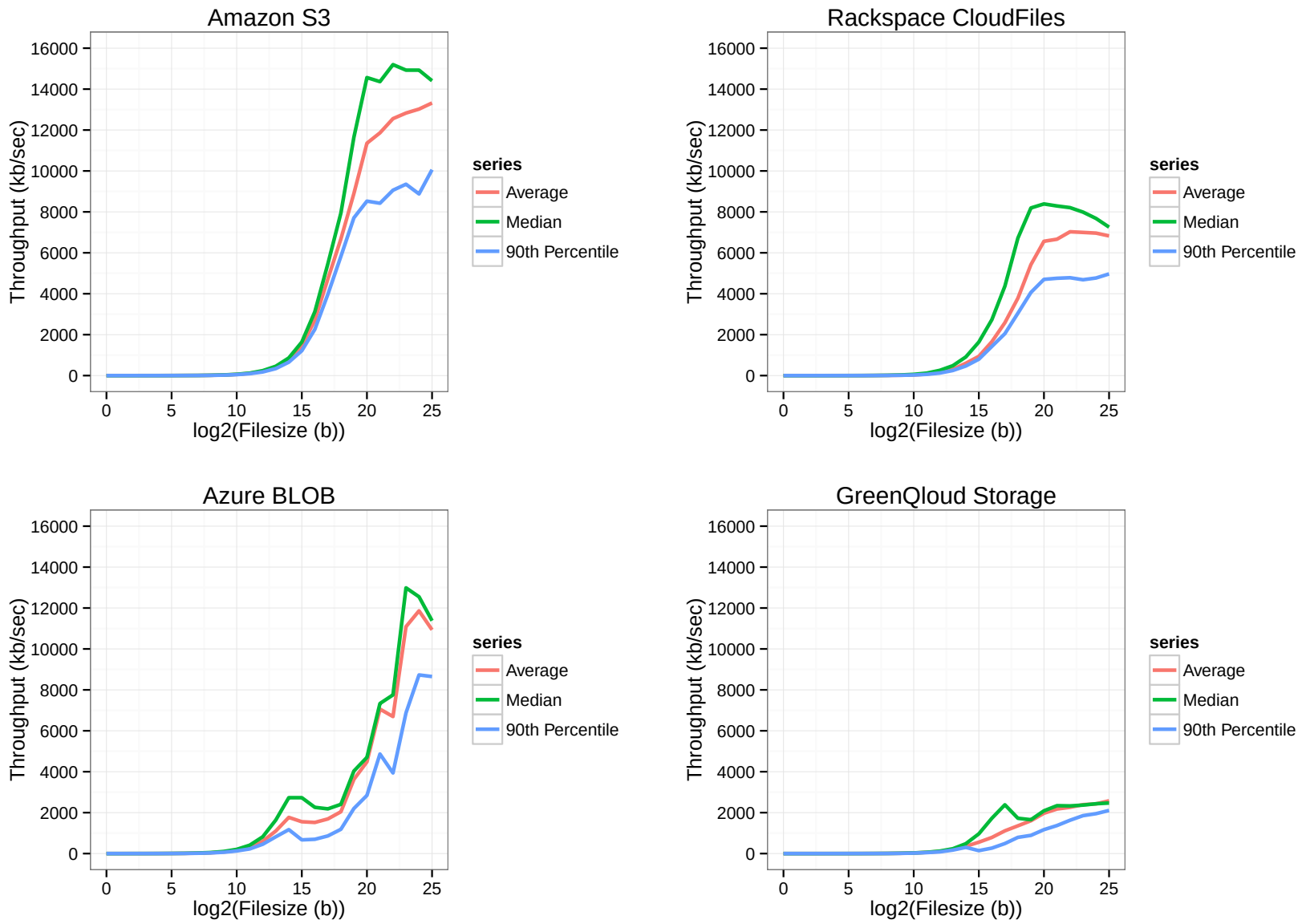


Figure 33: Throughput per thread for the HTTP GET verb.

When we look at the throughput of the PUT verb in Figure 34 (page 77) we see that Amazon takes a clear lead again, while RackSpace and Azure compete for the second place. GreenQloud is again the worst performing cloud provider, as it is not even possible to get over 2 MegaBytes per second per thread.

Both Amazon and Windows Azure are very competitive with regards to the DELETE verb, as shown in Figure 35 (page 78). They are also very fast, most responses are under 30 milliseconds. RackSpace is a lot slower but still manages an acceptable response time with the 90th percentile around 250 milliseconds. GreenQloud is again the slowest by a large margin. Most of the time the 90th percentile is above 1 second, only to drop when we reach larger file sizes. We have no explanation for this strange behavior.

Finally, we take a look at the response time for the LIST verb in Figure 36 (page 79). Strange enough, GreenQloud performs very well in this regard with the 90th percentile, median, and average around the 250 milliseconds mark. It also performs a lot more reliable than Windows Azure, which only has the median around 250 milliseconds. Rackspace performs slightly slower than Windows Azure, but does perform more predictable than Windows Azure. Amazon is the slowest when responding to the LIST verb.

Amazon also shows some strange behavior. Initially the responses decrease slightly as the file size increases. But once the files reach a size of 8 MiB, the 90th percentile response time increases to ± 2200 milliseconds. When the file size reaches 16 MiB, the 90th percentile response time increases to ± 8300 milliseconds. And when the file size reaches 32 MiB the 90th percentile increases to ± 13600 milliseconds. We think that this might have to do with distribution of files over multiple servers which have to be queried. But this is only an assumption, we do not know what causes this strange behavior.

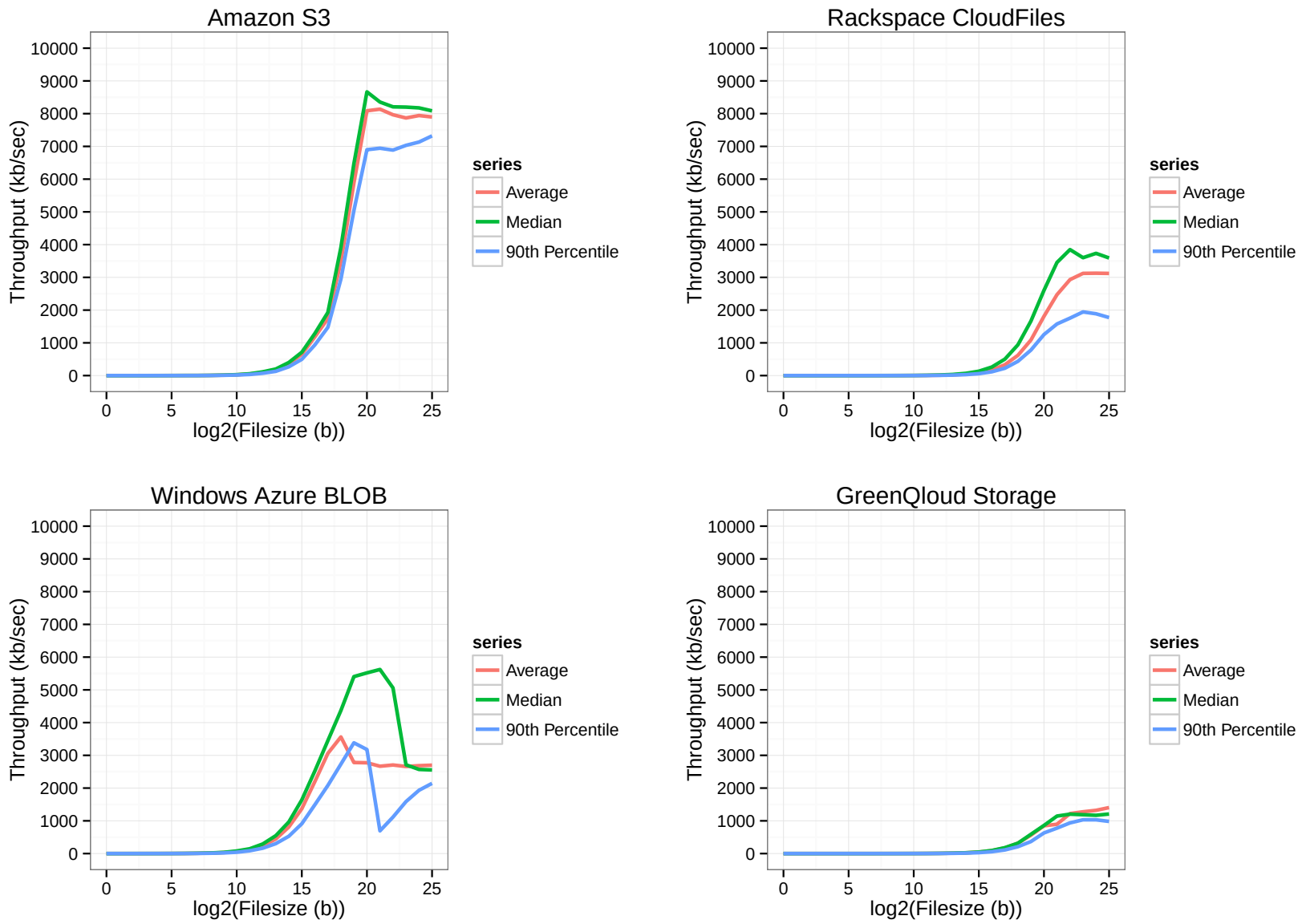


Figure 34: Throughput per thread for the HTTP PUT verb.

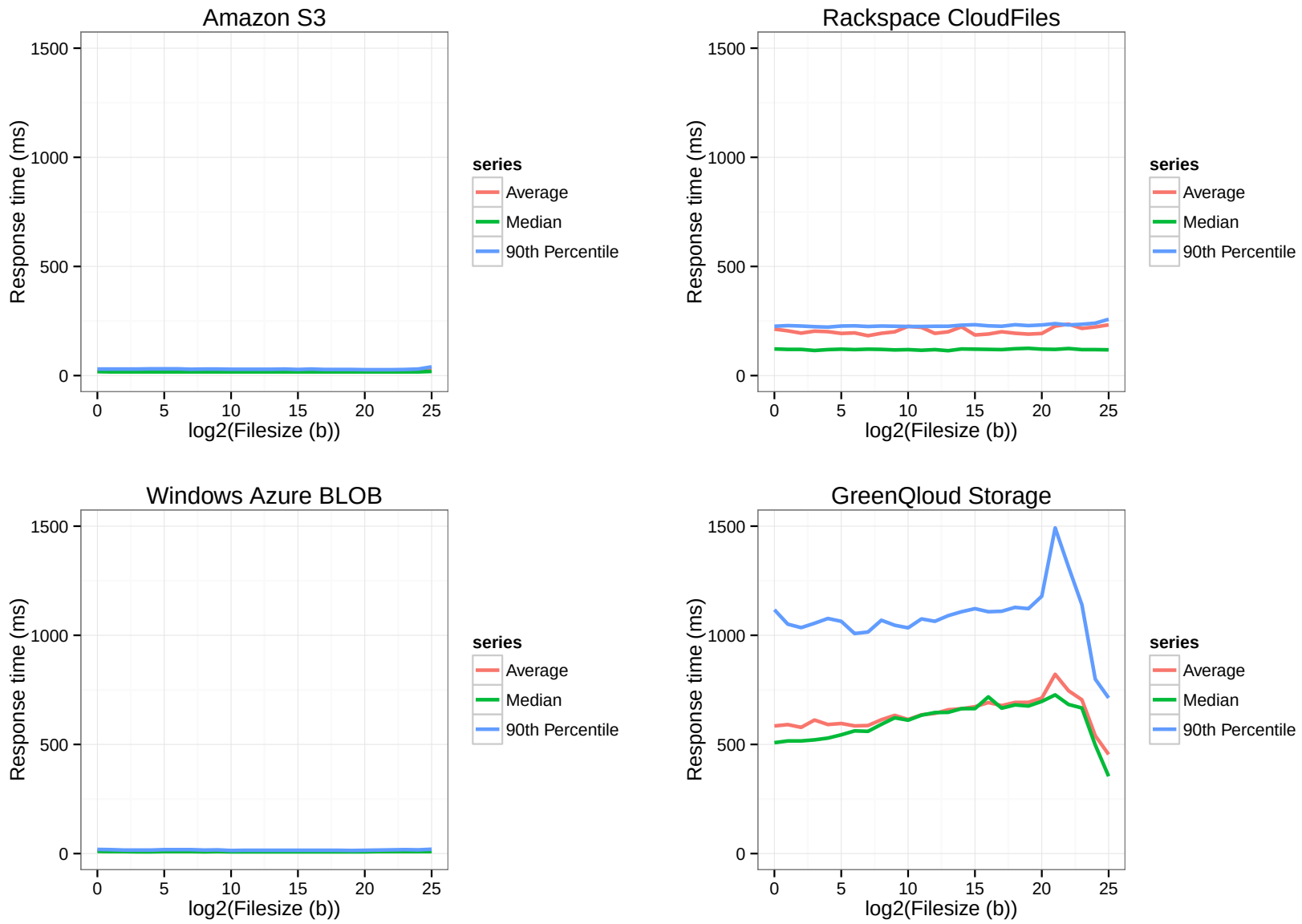


Figure 35: Response time for the HTTP DELETE verb.

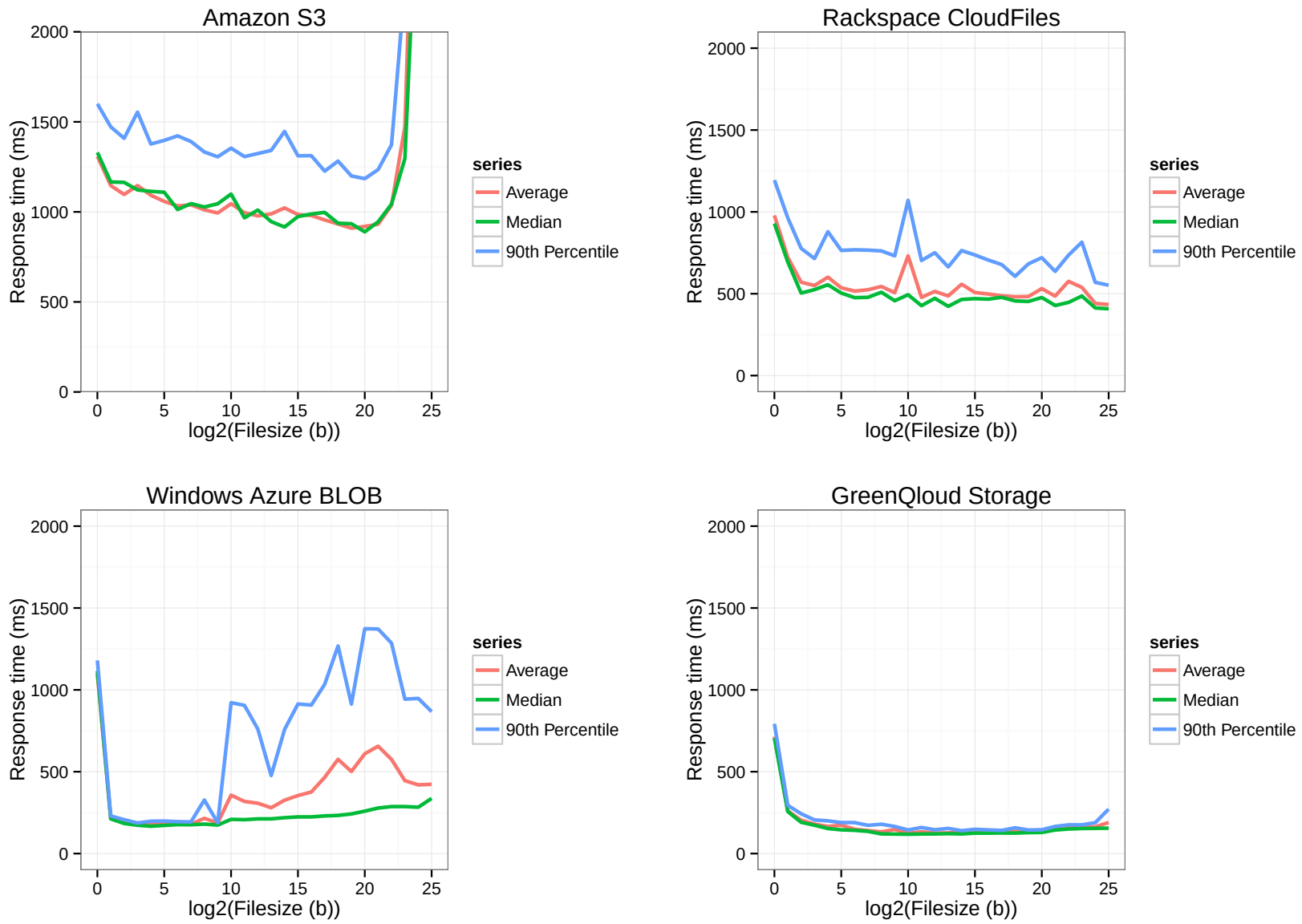


Figure 36: Response time for the HTTP LIST verb.

6.2 Streaming DepSky Local Performance Test

The architecture and design of the Streaming DepSky test is not that different from the first test setup. The only difference is that instead of simple using the `BlobStore` provided by JClouds, we use the `IFileReader` and `IFileWriter` interfaces exposed by the Streaming DepSky library. The data was gathered between June 14, 2013 and June 21, 2013. It should be noted that these test took an extremely long time per run, thus not as many data points as for the cloud tests are collected.

The availability statistics for both algorithms are shown in Table 7. It is clear that our implementation is not 100% Byzantine fault tolerant. However, for our intended use case this is a major improvement over using a single cloud provider. 96 % of the errors occured during a GET request. Since we focus on providing an append only storage, these failures are not that significant.

The 2 PUT requests that failed for the Streaming DepSky-A algorithm are far more concerning. We do see however that the Streaming DepSky-CA algorithm does not exhibit a single failing PUT request. The 100% success rate on the DELETE and LIST requests comes as no surprise, since the individual clouds also exhibited this behavior (except GreenQloud). This is shown graphically in Figure 37 (page 81).

HTTP Verb	Algorithm	Measurements	Failures	Availability
GET	S-DepSky-A	185344	6	99,999 %
	S-DepSky-CA	133120	55	99,995 %
PUT	S-DepSky-A	185344	2	99,999 %
	S-DepSky-CA	133120	0	100 %
DELETE	S-DepSky-A	185344	0	100 %
	S-DepSky-CA	133120	0	100 %
LIST	S-DepSky-A	181	0	100 %
	S-DepSky-CA	130	0	100 %
Overall	S-DepSky-A	556213	8	99,9999 %
	S-DepSky-CA	399490	55	99,9998 %

Table 7: Availability statistics for the Streaming DepSky algorithms

From Figure 37 (page 81) we also see that the bulk of the failed request, again, are at the end of the file size range. Thus the bigger the file, the larger the chance that the transfer will fail. This was also observed when we ran our tests on the individual clouds.

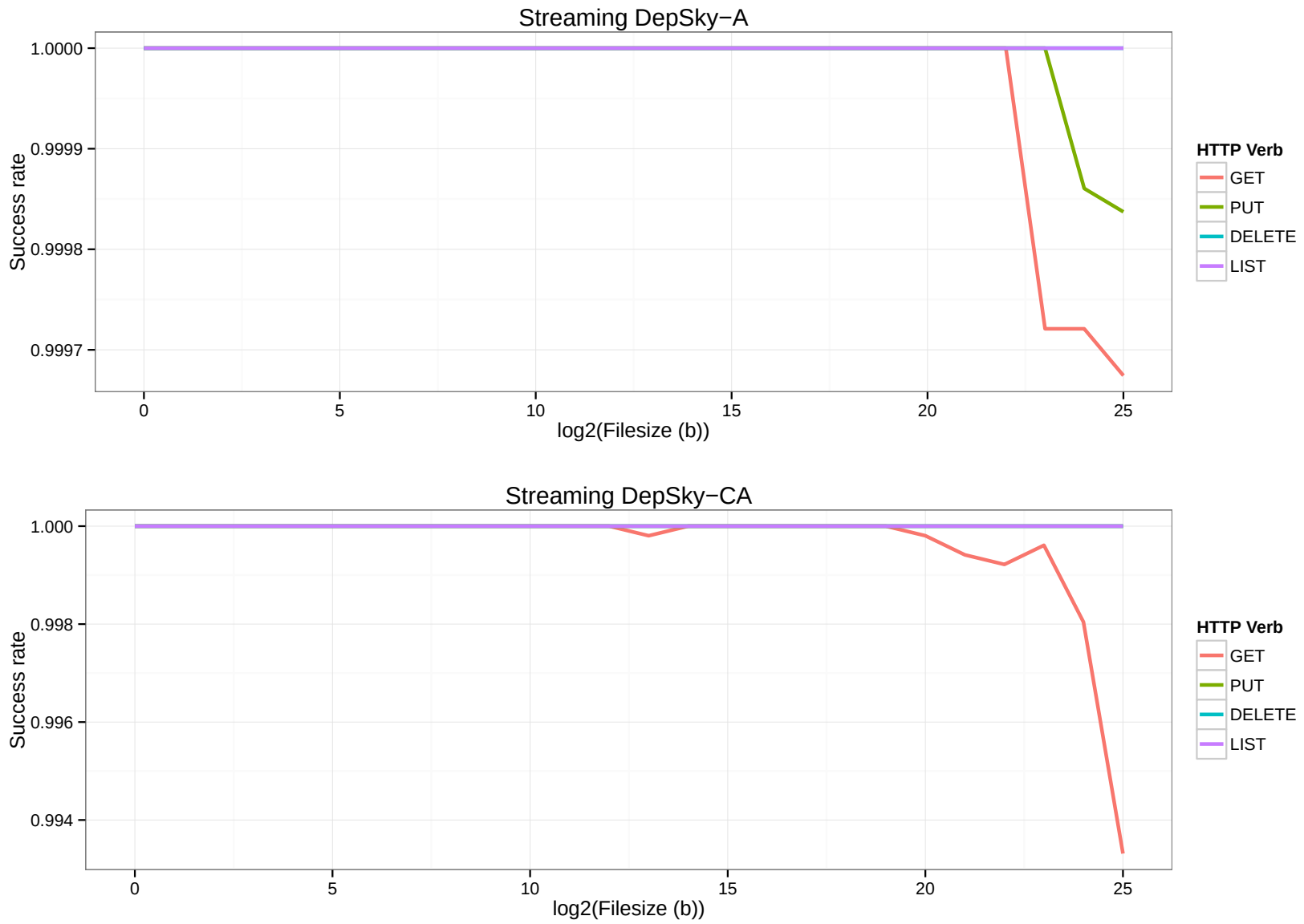


Figure 37: Availability of Streaming DepSky per algorithm per verb.

6.2.1 Streaming DepSky-A results

When we take a look at the throughput for the PUT request in Figure 38 (page 83), we see that the throughput is simply abysmal, we are only getting roughly 150 KiloBytes per second. However, this is easily explained when we consider that we only have a single 100 Megabit network connection. The test is run using 16 workers to upload 16 files concurrently. These 16 files are then uploaded to 4 clouds, thus there are 64 concurrent uploads. And if we calculate $150 \times 64 = 9600$, we see that we are simply saturating the network connection.

The same should not be true for the GET request, as the implementation will only read from one stream unless that stream fails. However, we see that the GET performance is almost identical. And indeed upon inspection of the bandwidth consumed, we noticed that the amount of data that was uploaded was equal to the amount of data that was downloaded! This was a very strange observation as multiple verifications of the code still did not reveal any way in which this could occur. We think that the Java HotSpot compiler might have played a role in this strange behavior, as it could have unrolled the loop in such a way that the application did not work 100% as intended. However, we were unable to verify this.

We also see that the DELETE and LIST request are much slower than the sum of those of the individual clouds. However this is easily explained, as both requests are now returning twice as much data. This is because of the uploading of a file and a metadata file. Thus instead of a single DELETE request, we now have to issue two DELETE requests. On top of that, some cloud providers maintain a size limit on the amount of items returned with a single LIST command. Thus listing a directory might take multiple requests.

6.2.2 Streaming DepSky-CA results

When we look at the performance measurements for the Streaming DepSky-CA algorithm we see what we expected to see. Since due to the erasure coding the size of the files uploaded is reduced by a factor $\frac{3f+1}{f+1} = \frac{4}{2} = 2$, the throughput increased by the same factor. We see this effect in both the GET and the PUT verb. Again, the same strange performance issue is observed as with the Streaming DepSky-A algorithm. This should not come as a suprise, as most components are shared between the algorithms.

It does appear that the performance of the DELETE command is much better with the Streaming DepSky-CA algorithm. But this is not really the case, as the results for the Streaming DepSky-A algorithm seem skewed by the large increase to about 6000 milliseconds. Since the only difference is the file size, and during our individual

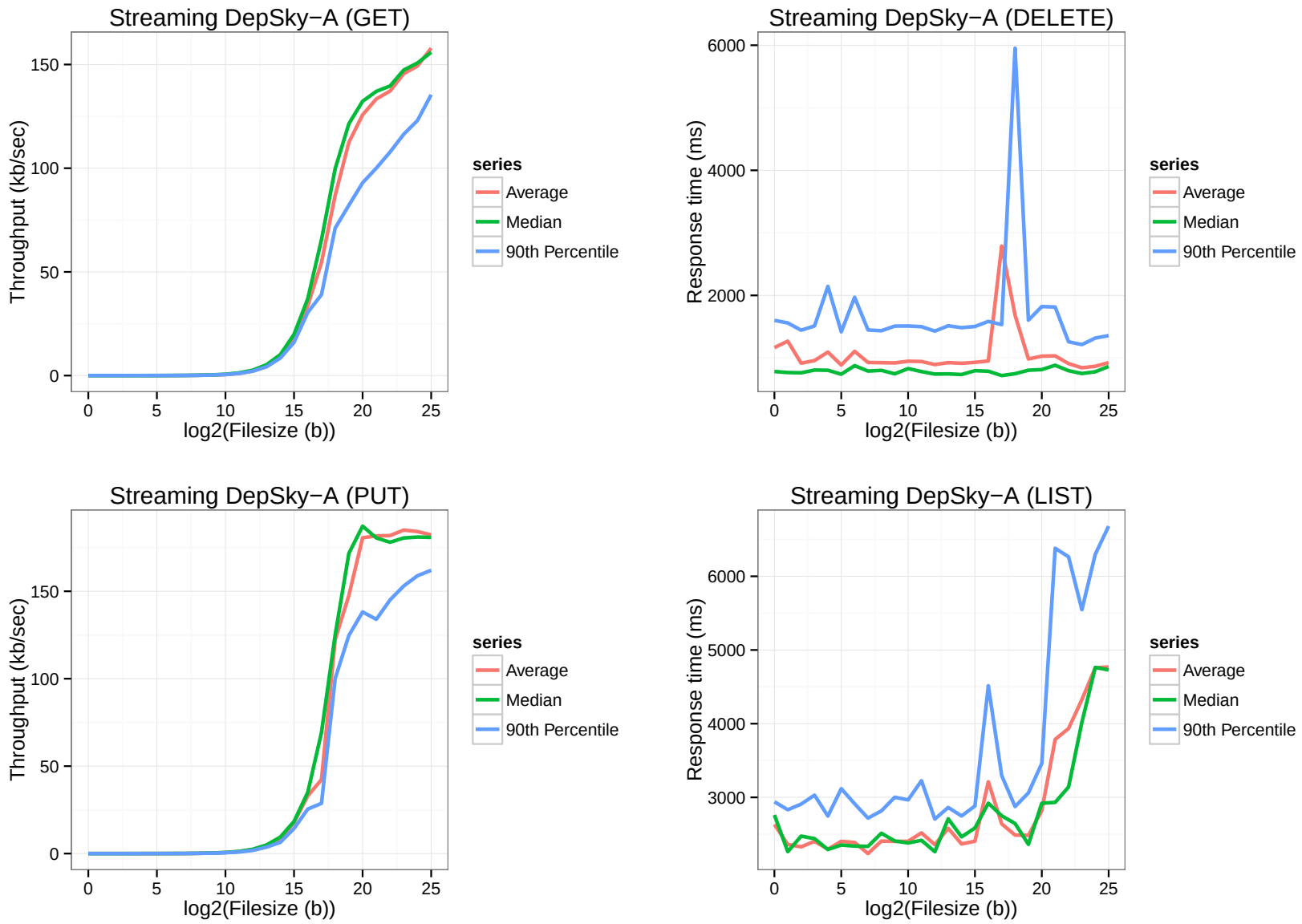


Figure 38: Performance of Streaming DepSky-A per verb.

cloud tests we did not see any significant correlation between file size and response time for the DELETE verb, we attest the peak in the DELETE verb graph in Figure 38 (page 83) to variance.

There was however a correlation between the response time of the LIST verb and the file size for at least Amazon and Azure. Thus since the files stored with the Streaming DepSky-CA algorithm are smaller, the response time of the LIST verb is lower.

6.3 Streaming DepSky Cloud Performance Test

Our final test was done in each cloud. This allows us to see if certain clouds provide better connectivity than others. For example, one cloud provider might send the data over the quickest path, while another might send the data over the most economical path. Since we already tested our algorithms with a large data set, we will use a smaller data set for these tests. We will only use the file sizes 2^{22} and 2^{23} or 4 and 8 MiB respectively. Since it was not possible at the time the benchmark was taken to create Virtual Machines at GreenQloud, there are no benchmarks that were run on a GreenQloud virtual machine. The GreenQloud storage was still available so the test setup did not have to be adjusted.

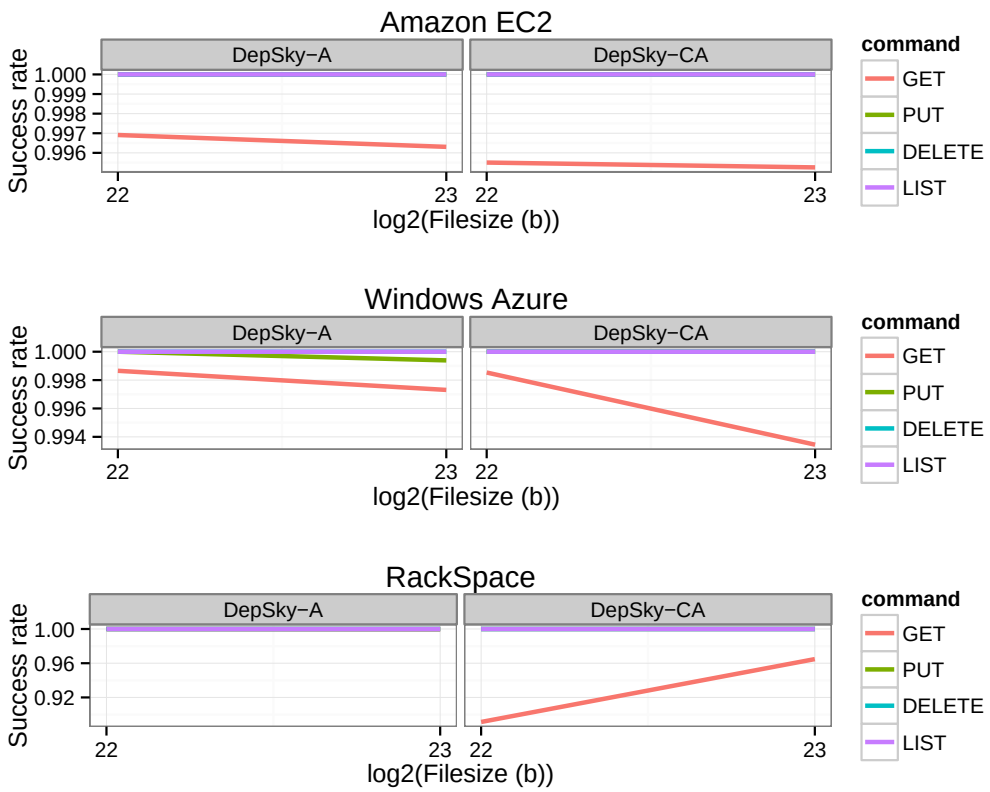


Figure 39: Perceived availability for the Streaming DepSky algorithms in the cloud.

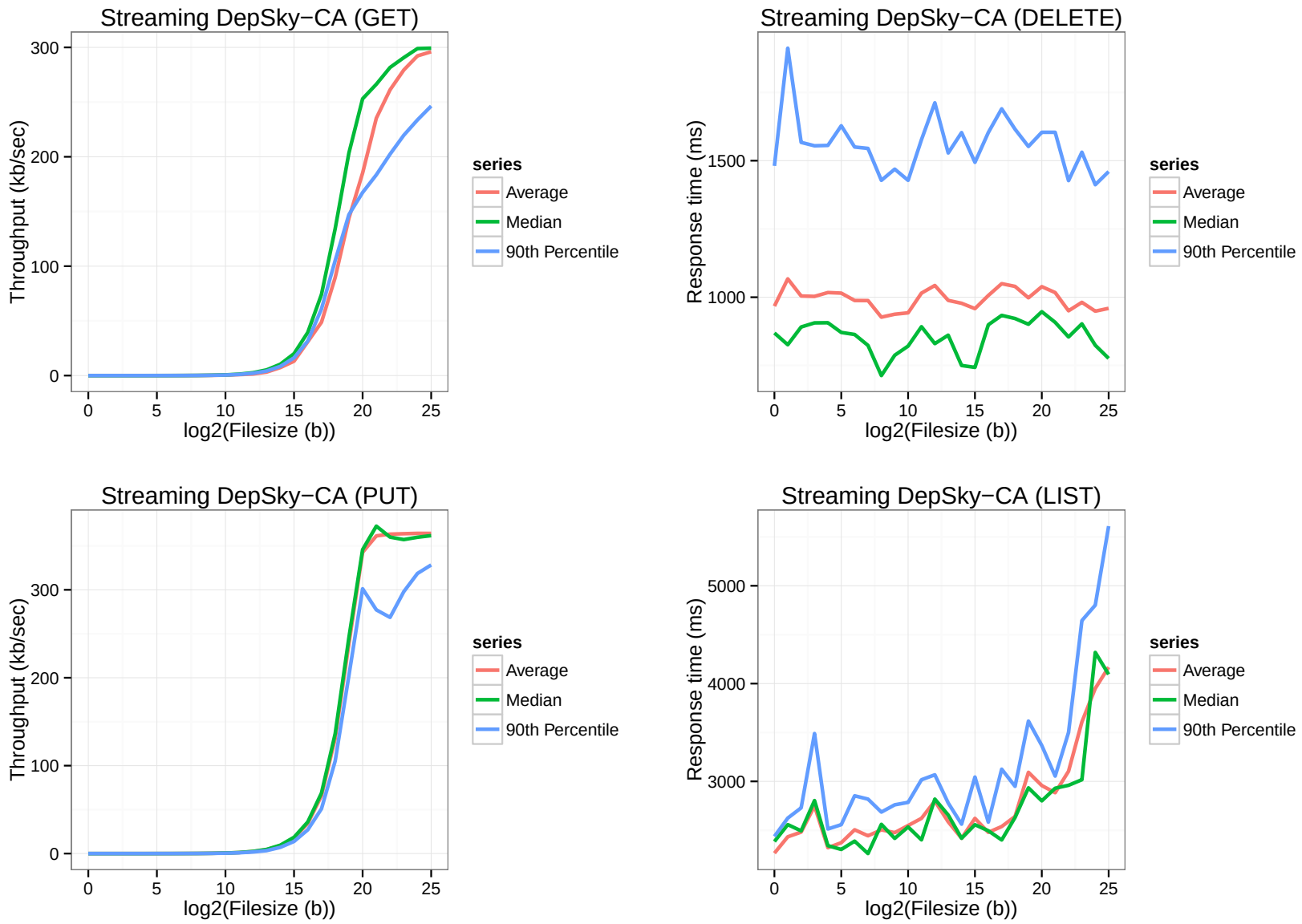


Figure 40: Performance of Streaming DepSky-CA per verb.

When we take a look at the availability in Figure 39 (page 84), we see that in almost all cases only the GET requests failed. Only on Windows Azure we noticed that PUT requests failed. The DELETE and LIST requests always succeeded, which is in line with our expectations.

If we look at the throughput of the GET requests, we see that without the 100 MegaBit bandwidth limit we are able to achieve much higher throughput. We see that for some reason, Amazon is much faster than Azure and RackSpace. The difference with RackSpace is explainable as its physical location is in the United States, while all the other cloud providers are located in Western Europe. Thus, the data transfered from RackSpace has to travel a much longer distance before it arrives. Clearly, the latency is then also a lot higher. And since we wait until we receive $n - f$ acks, RackSpace simply has to wait longer. However, this does not explain the

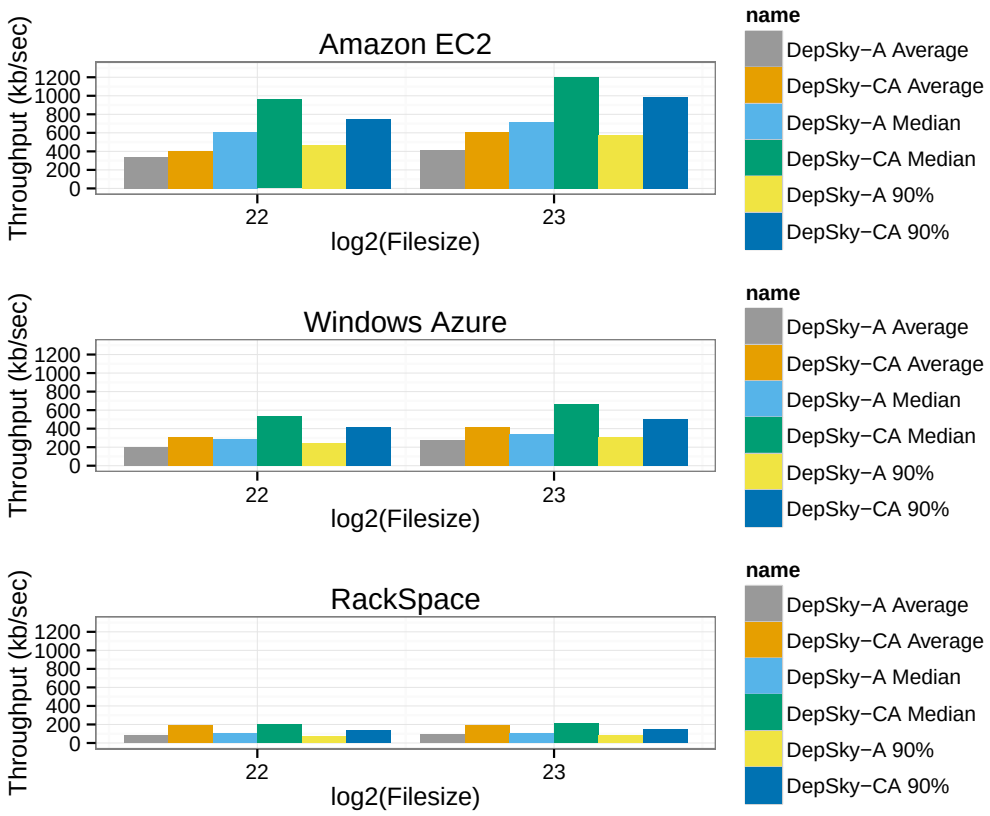


Figure 41: Throughput of the GET verb with the Streaming DepSky algorithms in the cloud.

difference between Windows Azure and Amazon. Physically, Windows Azure is located in Amsterdam and Amazon in Ireland. It might be the case that Amazon uses faster, and potentially more expensive, network routes to transmit the data. It is also possible that the connection from Ireland to Iceland is much faster, or more direct than from Amsterdam to Iceland. We expected that the GET performance would be roughly equal for both Azure and Amazon, so this is a surprising result.

When we look at the throughput for the PUT requests in Figure 42 (page 87), we see that this is even higher than the GET requests. This is strange, since we noticed that when we were testing the individual clouds that the throughput of the GET requests was faster than that of the PUT requests (Figures 33 and 34). It could be that the network connections for incoming data are faster than those for outgoing data. Or that the retrieval of data from the cloud providers is slower than the storing of data. But it remains a strange phenomenon none the less.

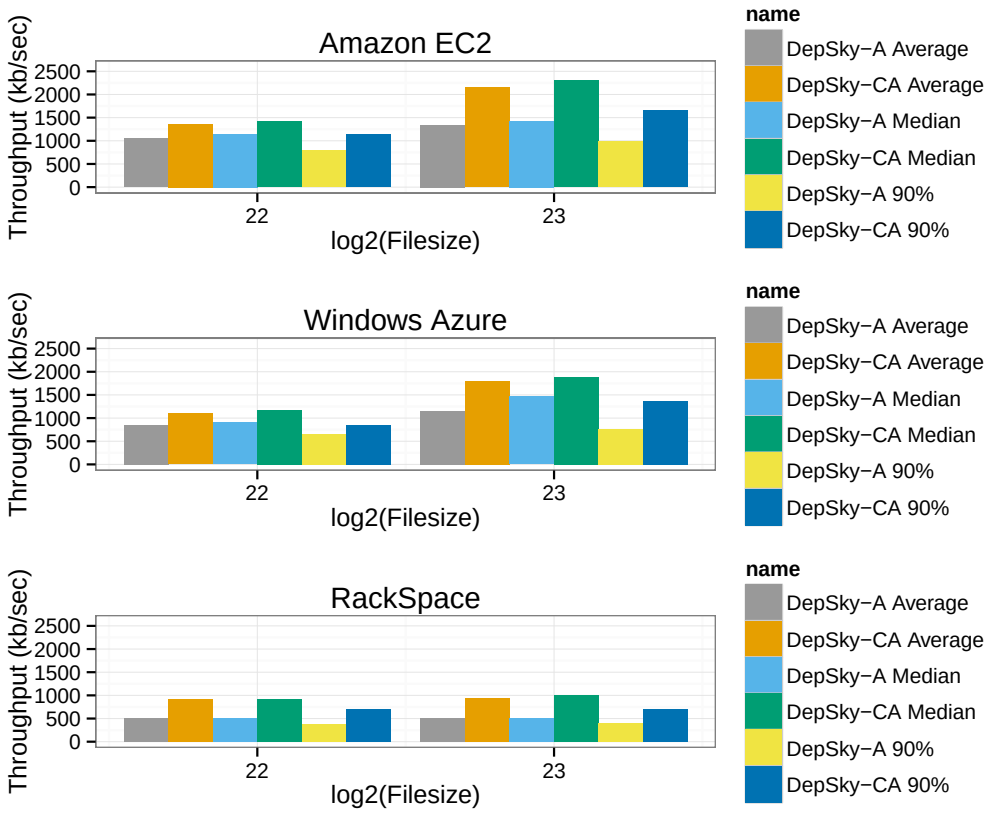


Figure 42: Throughput of the PUT verb with the Streaming DepSky algorithms in the cloud.

When we look at the response time of the DELETE requests in Figure 43 (page 88), we see that Azure and Rackspace are quite fast, while Amazon is a bit slower. Overall, the response times are still quite high. We would expect that the latency of the requests would have improved, assuming that the data centers in which the servers are hosted are connected with less hops to a major internet backbone than the TNO network. However, this does not seem to make any difference as the results are in line with what we saw when the application was running on the TNO network.

Using Amazon the LIST requests are executed faster than when the tests were run at TNO, as we can see in Figure 44 (page 89). While Windows Azure performs roughly

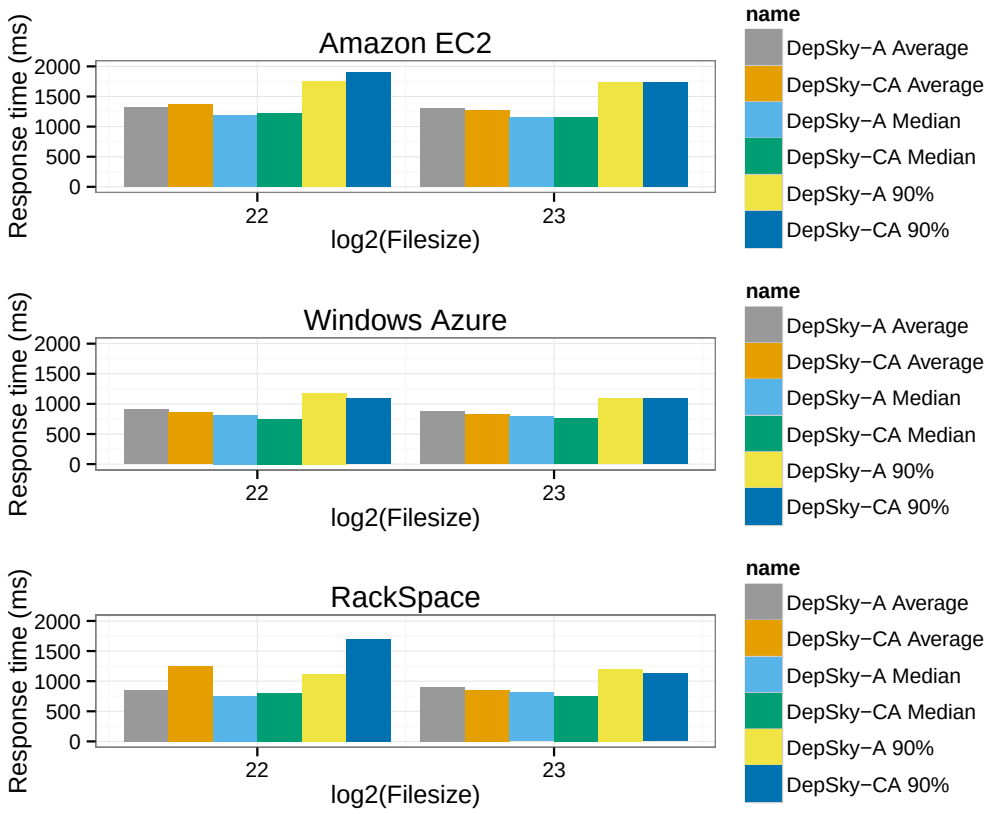


Figure 43: Response time of the DELETE verb with the Streaming DepSky algorithms in the cloud.

the same as when hosted at TNO, and Rackspace is significantly slower.

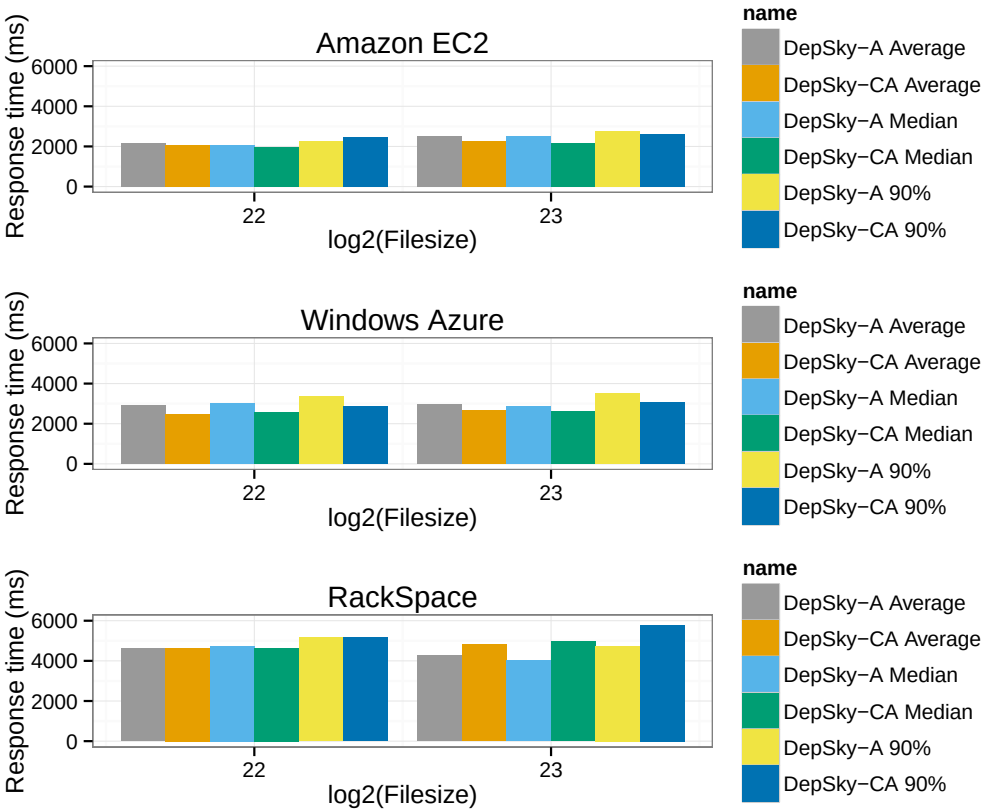


Figure 44: Response time of the LIST verb with the Streaming DepSky algorithms in the cloud.

7 Conclusion

We discussed the analysis and design of two improved storage algorithms. During the implementation we saw that large components can be shared between the algorithms, making the implementation of both of them cheaper. Using the documentation provided in this thesis, as well as components made available, it should not be difficult to build a production ready system.

During the performance evaluation we saw that while we were not able to achieve a 100% available system, we did vastly improve on the availability of the used cloud providers. The sum of using them all is greater than the parts. The most important use case we envisioned for our system was append only storage, and during our tests we attained a 100% success rate on file uploads using the Streaming DepSky-CA algorithm. Since the processing of the data is done in batches, having a file download fail is not that big a failure as it can easily be retried. Having a file upload fail is a much larger problem, since that would cause our data storage platform to retry the requests. This could lead to a significant bottleneck and potentially cause the system to crash as it is flooded with requests which it can not handle.

Our performance tests also showed that we were able to upload roughly 20 MegaBytes per second (90 percentile), using 16 threads and the Streaming DepSky-CA algorithm. Thus, if we consider the data load analysis in Section 1.1 (page 2), we are able to process the data generated by a sensor network of 10 000 nodes that each submit up to 50 measurements per second. This is without considering compression techniques that can be applied before uploading.

7.1 Costs of Implementation

We assume a data set of 150 TeraBytes (TB) and $n = 4$ thus $f = 1$. For the Streaming DepSky-A algorithm we store 150 TB at each cloud provider, and for the Streaming DepSky-CA we store 75 TB at each provider. The prices of storing data at a provider are given in Table 8, and the cost of storing the data set are given in Table 9.

We see that the cost of storing 1 TeraByte of data for one year is (roughly) $ 3943 using the DepSky-A algorithm and $ 2023 when using the DepSky-CA algorithm. Given that this data is distributed geographically across 2 continents, and the storage solution is highly fault tolerant, this does not seem to be very expensive when compared with traditional enterprise storage.

We also expect that the cloud providers will continue to lower the costs of storing data in the coming years. Thus the total price of storing the data set will continue to decrease.

Cloud Provider	Data Upto	Price ($) GB
Amazon S3	1 TB	$ 0,095
	50 TB	$ 0,08
	450 TB	$ 0,07
Windows Azure	1 TB	$ 0,095
	50 TB	$ 0,08
	450 TB	$ 0,07
RackSpace	1 TB	$ 0,10
	50 TB	$ 0,09
	200 TB	$ 0,085
GreenQloud	–	$ 0,095

Table 8: Storage costs per provider

	DepSky-A	DepSky-CA
Amazon S3	$ 11015	$ 5765
Windows Azure	$ 11015	$ 5765
RackSpace	$ 13010	$ 6635
GreenQloud	$ 14250	$ 7125
Total / month	$ 49290	$ 25290
Total / year	$ 591480	$ 303480
Price / TB / year	$ 3943,20	$ 2023,20

Table 9: Costs of storing the dataset (150 TB)

References

[1] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "Depsky: dependable and secure storage in a cloud-of-clouds," in *Proceedings of the sixth conference on Computer systems*, ser. EuroSys '11. ACM, 2011, pp. 31–46. [Online]. Available: http://www.gsd.inesc-id.pt/~mpc/pubs/eurosys219-bessani.pdf

[2] F. Zhao and L. Guibas, *Wireless Sensor Networks: An Information Processing Approach*, ser. Electronics & Electrical. Elsevier/Morgan Kaufmann Publishers, 2004. [Online]. Available: http://books.google.nl/books?id=BkaQkhkWGfoC

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672

[4] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, "Object storage: the future building block for storage systems," in *Local to Global Data Interoperability - Challenges and Technologies, 2005*, 2005, pp. 119–123. [Online]. Available: https://researchweb.watson.ibm.com/haifa/projects/storage/objectstore/papers/PositionOSD.pdf

[5] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[6] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, 2010, pp. 1–10.

[7] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 26, no. 1, pp. 64–69, Jan. 1983. [Online]. Available: http://doi.acm.org/10.1145/357980.358007

[8] N. Marz and J. Warren, *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications, 2013. [Online]. Available: http://www.manning.com/marz/

[9] J. van der Veen, B. van der Waaij, and R. Meijer, "Sensor data storage performance: Sql or nosql, physical or virtual," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, 2012, pp. 431–438.

[10] E. Eliasson and D. Öhrlund, "Scalable storage of sensor data for a high frequency measurement vehicular monitoring system," Master's thesis, Luleå University of Technology, 2012. [Online]. Available: http://pure.ltu.se/portal/files/41575378/LTU-EX-2012-41494352.pdf

[11] B. Pernici, M. Aiello, J. vom Brocke, B. Donnellan, E. Gelenbe, and M. Kretsis, "What is can do for environmental sustainability: a report from caise 2011 panel on green and sustainable is," *Communications of the Association for Information Systems*, vol. 30, no. 1, p. 22, 2012. [Online]. Available: http://eprints.nuim.ie/3021/1/CAiSE_Panel_Green_IS-Dec13.pdf

[12] J. Wu, L. Ping, X. Ge, Y. Wang, and J. Fu, "Cloud storage as the infrastructure of cloud computing," in *Intelligent Computing and Cognitive Informatics (ICICCI), 2010 International Conference on*, 2010, pp. 380–383. [Online]. Available: https://wiki.csc.calpoly.edu/csc560/raw-attachment/wiki/Meth_Bibliography/cloudStorage.pdf

[13] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (raid)," *SIGMOD Rec.*, vol. 17, no. 3, pp. 109–116, Jun. 1988. [Online]. Available: http://doi.acm.org/10.1145/971701.50214

[14] G. Zhao, M. Jaatun, A. Vasilakos, A. Nyre, S. Alapnes, Q. Yue, and Y. Tang, "Deliverance from trust through a redundant array of independent net-storages in cloud computing," in *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*, 2011, pp. 625–630.

[15] K. D. Bowers, A. Juels, and A. Oprea, "Hail: a high-availability and integrity layer for cloud storage," in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 187–198. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653686

[16] M. Schnjakin, R. Alnemr, and C. Meinel, "A security and high-availability layer for cloud storage," in *Web Information Systems Engineering – WISE 2010 Workshops*, ser. Lecture Notes in Computer Science, D. Chiu, L. Bellatreche, H. Sasaki, H.-f. Leung, S.-C. Cheung, H. Hu, and J. Shao, Eds. Springer Berlin Heidelberg, 2011, vol. 6724, pp. 449–462. [Online]. Available: http://www.hpi.uni-potsdam.de/fileadmin/hpi/FG_ITS/papers/Trust_and_Security_Engineering/2010_Schnjakin_CISE.pdf

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

[18] C. Baun and M. Kunze, "Redundant cloud storage with octopus," -, vol. -, pp. –, 2011.

[19] J. Spillner, J. Müller, and A. Schill, "Creating optimal cloud storage systems," *Future Generation Computer Systems*, vol. 29, no. 4, pp. 1062 – 1072, 2013, <ce:title>Special Section: Utility and Cloud Computing</ce:title>. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X12001409

[20] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, "Racs: a case for cloud storage diversity," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10, 2010, pp. 229–240. [Online]. Available: http://www.cs.cornell.edu/projects/racs/pubs/racs-socc2010.pdf

[21] Y. Singh, F. Kandah, and W. Zhang, "A secured cost-effective multi-cloud storage in cloud computing," in *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*, 2011, pp. 619–624.

[22] C. Cachin, R. Haas, and M. Vukolic, "Dependable storage in the intercloud," *IBM Research*, vol. 3783, pp. 1–6, 2010.

[23] B. Cohen, "The bittorrent protocol specification," 06 2009. [Online]. Available: http://www.bittorrent.org/beps/bep_0003.html

[24] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982. [Online]. Available: http://doi.acm.org/10.1145/357172.357176

[25] Microsoft, *Understanding Block Blobs and Page Blobs*, Microsoft, 6 2012. [Online]. Available: http://msdn.microsoft.com/en-us/library/windowsazure/ee691964.aspx

[26] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, Nov. 1979. [Online]. Available: http://doi.acm.org/10.1145/359168.359176

[27] B. Kaliski, "Twirl and rsa key size," RSA Laboratories, Tech. Rep., 5 2003. [Online]. Available: http://www.rsa.com/rsalabs/node.asp?id=2004#st03

[28] N. Ferguson and B. Schneier, *Practical cryptography*. Wiley, 2003. [Online]. Available: http://books.google.nl/books?id=7SiKtxPrrRMC

[29] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2," University of Tennessee, Tech. Rep. CS-08-627, August 2008. [Online]. Available: http://www.cs.utk.edu/~plank/plank/papers/CS-08-627.html

[30] J. S. Plank, E. L. Miller, and W. B. Houston, "GF-Complete: A comprehensive open source library for Galois Field arithmetic," University of Tennessee, Tech. Rep. UT-CS-13-703, January 2013. [Online]. Available: http://web.eecs.utk.edu/~plank/plank/papers/CS-13-703.html

[31] B. Kaliski, "Pkcs# 5: Password-based cryptography specification version 2.0," RSA Laboratories, Tech. Rep., 2000.

[32] ——, "Pkcs# 7: Cryptographic message syntax version 1.5," RSA Laboratories, Tech. Rep., 1998.

[33] M. S. Rasmussen, "Pushing the limits of amazon s3 upload performance," 11 2011, blog posting. [Online]. Available: http://improve.dk/pushing-the-limits-of-amazon-s3-upload-performance/

[34] Nasuni, "The state of cloud storage 2013 industry report," Nasuni, Tech. Rep., 2013.

[35] D. Parkhill, *The challenge of the computer utility*. Parkhill, D.F.Addison-Wesley Pub. Co., 1966.

[36] P. Mell and T. Grance, "The nist definition of cloud computing," National Institute of Standards and Technology, Information Technology Laboratory, Tech. Rep., Sep. 2011. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[37] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, ser. International computer science series. Addison-Wesley, 2011. [Online]. Available: http://books.google.nl/books?id=GmYpKQEACAAJ

[38] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[39] F. Mattern, “Virtual time and global states of distributed systems,” *Parallel and Distributed Algorithms*, vol. 1, no. 23, pp. 215–226, 1989.

[40] C. Fidge, “Logical time in distributed computing systems,” *Computer*, vol. 24, no. 8, pp. 28–33, 1991.

[41] M. Raynal and M. Singhal, “Logical time: capturing causality in distributed systems,” *Computer*, vol. 29, no. 2, pp. 49–56, 1996.

[42] V. Hadzilacos and S. Toueg, “A modular approach to fault-tolerant broadcasts and related problems,” 1994.

[43] N. Lynch, *Distributed Algorithms*, ser. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 1996. [Online]. Available: http://books.google.nl/books?id=2wsrLg-xBGgC

[44] M. Tanase, “Ip spoofing: an introduction,” *Security Focus*, vol. 11, 2003.

[45] T. Connolly and C. Begg, *Database Systems: A Practical Approach to Design, Implementation, and Management*, ser. International computer science series. Addison-Wesley Longman, Incorporated, 2005. [Online]. Available: http://books.google.nl/books?id=jJbnDxiJ4joC

[46] R. C. Martin. Principles of object oriented design. [Online]. Available: http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[47] R. Martin, *Agile Software Development: Principles, Patterns, and Practices*, ser. Alan Apt Series. Prentice Hall/Pearson Education, 2003. [Online]. Available: http://books.google.nl/books?id=0HYhAQAAIAAJ

[48] T. DeMarco, *Structured analysis and system specification*, ser. Prentice-Hall software series, 1979. [Online]. Available: http://books.google.nl/books?id=RmpRAAAAMAAJ

[49] B. Meyer, *Object-oriented software construction*, ser. Prentice-Hall international series in computer science. Prentice-Hall, 1988. [Online]. Available: http://books.google.nl/books?id=kn0xNyridvkC

[50] R. Johnson, E. Gamma, J. Vlissides, and R. Helm, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. [Online]. Available: http://books.google.nl/books?id=iyIvGGp2550C

Appendix

A1 Cloud Computing & Distributed Systems background

In this appendix we introduce and discuss fundamental concepts and techniques that are relevant for understanding the problem addressed in this thesis.

A1.1 Cloud Computing

In 1966 a Canadian technologist named Douglas Parkhill, wrote the book *The Challenge of the Computer Utility* [35]. In this book he described the characteristics of delivering computing as a utility, much like water, gas, electricity and telecommunications are delivered. He called it 'Utility Computing' and predicted that organizations would be able to acquire as much IT services as they needed, whenever and wherever they needed them. By applying Utility Computing there could be a concealment of the complexity of IT, reduction of operational expenses, and converting of IT costs to variable 'on-demand' services. Less than 40 years later this concept, now called Cloud Computing, is a commercial reality.

The accepted **N**ational **I**nstitute of **S**tandards and **T**echnology (NIST) definition of cloud computing is that it is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [36]. Along side this definition they also distinguish five essential characteristics, three service models, and four deployment models.

A1.1.1 Essential characteristics

The five essential characteristics of cloud computing defined by the NIST are:

- **On-demand self-service** – A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.

- **Broad network access** – Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).
- **Resource pooling** – The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth.
- **Rapid elasticity** – Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.
- **Measured service** – Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

A1.1.2 Service models

Infrastructure **a**s **a** **S**ervice (IaaS) is the most basic cloud service model. IaaS providers offer computers, as physical or more often as virtual machines, as well as other infrastructure essentials such as: raw (block) and file-based storage, firewalls, load balancers, IP addresses, etc. Software and operating systems are usually provided in the form of images that can be installed on these virtual or physical machines. Maintenance of everything except the machine hardware should be done by the user.

In the **P**latform **a**s **a** **S**ervice (PaaS) service model, cloud providers deliver a computing platform typically including operating system, programming language execution environment, database, and web server. Application developers can develop and run their software solutions on a cloud platform without the cost and complexity of buying and managing the underlying hardware and software layers. With some PaaS

offers, the underlying computer and storage resources scale automatically to match application demand such that cloud user does not have to allocate resources manually.

Finally, in the **S**oftware **a**s **a** **S**ervice (SaaS) service model, cloud providers install and operate application software in the cloud and cloud users access the software from cloud clients. The cloud users do not manage the cloud infrastructure and platform on which the application is running. This eliminates the need to install and run the application on the cloud user's own computers simplifying maintenance and support. What makes a cloud application different from other applications is its scalability. If the load on a cloud application increases, the system will automatically delegate work to other machines, or even start additional machines to increase the capacity of the system. This process is transparent to the cloud user who sees only a single access point.

A schematic representation of the three service models is shown in Figure 45 (page 98)[36]. Layers should be read top down, thus SaaS is above PaaS in this image. Each layer abstracts what is beneath it, but it is not a strict layering pattern. Software running on a PaaS implementation might be able to access features provided by the IaaS layer (if the cloud provider supports or allows this).

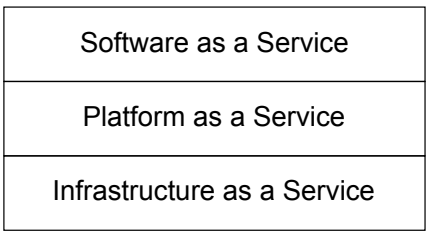


Figure 45: Hierarchy of cloud service models

A1.1.3 Deployment models

The most well known deployment model of a cloud is the *public cloud*. In this model the cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider. Examples of public cloud providers are: Amazon Web Services, Microsoft Windows Azure, Rackspace, etc.

[36]Inspired by `http://en.wikipedia.org/wiki/File:Cloud_computing_layers.png`

There is also the notion of a *private cloud*, in this model the cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises.

A specific form of a *private cloud* is the *community cloud*. In this model the cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.

Finally, it might also be possible that organizations that own a *private cloud* also use (parts of) a *public cloud*, this is called a *hybrid cloud*. In this case the cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability (e.g., 'Cloud bursting' for load balancing between clouds).

A schematic representation of the different deployment models can be seen in Figure 46 (page 99)[37]. Note that a community cloud is considered a private cloud in this image.

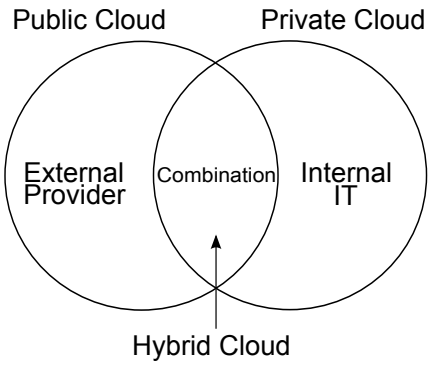


Figure 46: Venn diagram of the types of cloud computing

[37]Inspired by `http://netmediablog.com/types-of-cloud-computing/`

A1.2 Distributed system models

The purpose of the distributed system models is to make explicit all of the relevant assumptions about the system we are modeling. This allows us to make generalizations concerning what is possible and impossible, given those assumptions. These generalizations may take the form of general-purpose algorithms or desirable properties that are guaranteed. The guarantees are dependent on logical analysis and, where appropriate, mathematical proof. The aspects of distributed systems that are captured in these fundamental models are intended to help us discuss and reason about these systems their interaction, security and the ways in which they can fail. These are captured in a *interaction model*, a *security model*, and a *failure model* respectively.

We will introduce each model in a separate subsection for clarity, we will start with the interaction model in Section A1.2.1 (page 100), followed by the security model in Section A1.2.2 (page 104) and finally the failure model in Section A1.2.3 (page 107). The theory in this section and its subsections is largely based on [37].

A1.2.1 Interaction model

Computation occurs within processes, and processes interact by passing messages to each other. This results in communication (or information flow) and coordination (as in synchronization and ordering of activities) between these processes. The interaction model reflects the facts that communication takes place with delays that can be of considerable duration, and that the accuracy with which the independent processes can be coordinated is limited by these delays. It also reflects the fact that it is very difficult to maintain the same notion of time across all the computers in a distributed system.

As opposed to programs that run in a single process and follow a algorithm in which the steps to take are strictly sequential and the corresponding state is determined by these steps, distributed systems are composed of multiple processes that can be distributed over various machines. Their behavior and state should thus be described by a *distributed algorithm*, which is a definition of the steps to be taken by each of the processes of which the system is composed, as well as the transmission of messages between these processes. These messages are used to transfer information and coordinate the activity that they are performing. The rate at which each process proceeds and the timing of the transmission of messages between them can, in general, not be predicted. Because a distributed system can also fail at a process or communication channel level, it is also difficult to describe all the states of a dis-

tributed algorithm.

These processes, and their interaction, perform all of the activity of a distributed system. Each process has its own state, consisting of a set of data and variables that it can access and update. Since the state of each process is strictly private, it cannot be accessed or updated by any other process. The most significant factors affecting interacting processes in a distributed system are *communication performance*, the *impossibility of a single global notion of time*, and *event ordering*.

Performance of communication channels Since the communication between processes will generally be over a computer network, we identify the following performance characteristics:

- Latency

  This is the time difference between the starting of a transmission (t_{start}) and the start of receipt (r_{start}). Logically, latency (l) is thus $l = |r_{start} - t_{start}|$.

- Bandwidth

  This is the amount of information that can be transmitted over a channel within a time unit. Common bandwidth indicators are bits per second, or kilobytes per second.

- Jitter

  This is the difference in time needed to transmit a series of messages. Thus if we have two batches of messages (b_1 and b_2), a function which returns the amount of time taken to send a single message ($f(x)$) and the function that returns the time taken to send a batch of messages ($F(b) = \sum_{i=1}^{n} f(b_i)$). Jitter ($j$) can be defined as $j = |F(b_1) - F(b_2)|$.

Obviously these characteristics can vary over time, due to network routing, load, failures, etc.

Time and timing events Each computer in a distributed system has its own internal clock, this clock is used by local processes to obtain the value of the current time. This allows two processes each running in a different computer system to associate timestamps to their generated events. However, even if the two processes each read their clocks at the same time, their local clocks may supply different time values. This is because computer clocks drift from the perfect time and, more importantly, their drift rates differ from one another.

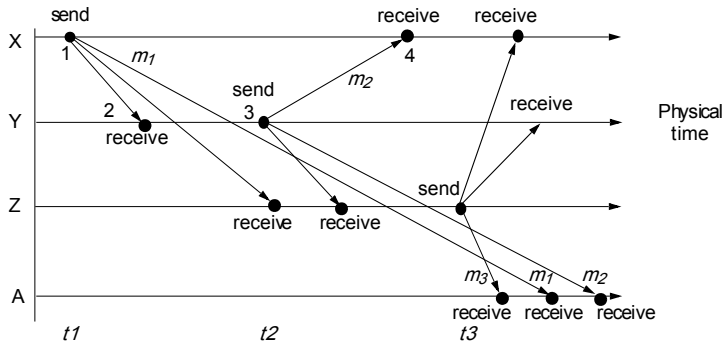


Figure 47: Real time ordering of events [37]

This *clock drift rate* is the rate at which the computer clock deviates from the perfect reference clock. Even if all the clocks in a distributed system are set to the exact same time, their clocks will eventually vary quite significantly unless corrections are applied. These corrections are known as clock synchronization, for example using a radio receiver to get time readings from the **G**lobal **P**ositioning **S**ystem (GPS) with an accuracy of about 1 microsecond. However GPS receivers do not work reliably inside buildings, nor is the cost justified for every computer. An alternative is to use the **N**etwork **T**ime **P**rotocol (NTP) to synchronize the clocks of the computers with another computer that has a reliable time source such as GPS, which has an accuracy of about 1 millisecond. The resulting agreement between times on the local clocks is obviously affected by variable message delays.

Event ordering In many cases, we are interested in knowing whether an event (e.g. sending or receiving a message) at one process occurred before, after or concurrently (at the same time) with another event at another process. The execution of a system can be described in terms of events and their ordering despite the lack of accurate clocks. For example, in Figure 47 (page 102) processes X, Y, Z, and A are exchanging messages. At t_1 process X broadcasts message m_1, at time 2 this message is received by process Y, in turn Y broadcasts its reply m_2. The third process Z broadcasts m_3 after having received both m_1 and m_2. However, due to network latency or invalid clock synchronization process A perceives a different order of messages, namely m_3, m_1, m_2. This could cause process A to behave differently. Thankfully we can avoid this from occurring by moving away from physical time and using a virtual time representation, as proposed in [38, 39, 40, 41][38].

Now we know what the problems are with the interaction between components of a distributed system, we can model them. Because it is hard to set limits on the time that can be taken for process execution, message delivery or clock drift, two oppos-

[38]As an example, consider a increasing sequence number on each message sent by a process. This allows for a total order on the messages sent by a single process.

ing positions provide a pair of simple models: *synchronous distributed systems* and *asynchronous distributed systems*. The first makes strong assumptions about time, while the latter makes no assumptions about time.

Synchronous distributed systems A synchronous distributed system is defined [42] as a distributed system where the following bounds are known:

- The time to execute each step of a process has known lower and upper bounds.
- Each message transmitted over a channel is received within a known bounded time.
- Each process has a local clock whose drift rate from real time has a known bound.

While it is possible to suggest likely upper and lower bounds for process execution time, message delay, and clock drift rates, it is difficult to arrive at realistic values and to provide guarantees of the chosen values. Unless the values of the bounds can be guaranteed, any design based on the chosen values will not be reliable [37]. However, modeling an algorithm as a synchronous system may be useful for giving some idea of how it will behave in a real distributed system.

That said, synchronous distributed systems can be built. What is required is for the processes to perform tasks with known resource requirements for which they can be guaranteed sufficient processor cycles and network capacity, and for processes to be supplied with clocks with bounded drift rates.

Asynchronous distributed systems As opposed to a synchronous distributed system, an asynchronous distributed system is one in which there are no bounds on[43]:

- Process execution speed – one process step may take a picosecond, while another takes a century; the time taken can be arbitrarily long.
- Message transmission delays – one message from process A to B may be delivered in negligible time and another may take several years; the time taken can thus be arbitrarily long.
- Clock drift rates – the same applies here; clocks can drift arbitrarily.

The asynchronous model allows no assumptions about the time intervals involved in any execution. This is an exact model of the Internet, where there is no intrinsic bound on server or network load. However, some design problems can be solved even with these assumptions.

A1.2.2 Security model

The modular nature of distributed systems and their openness exposes them to attack by both external and internal agents. The security of a distributed system can be achieved by securing the processes and the channels used for their interactions and by protecting the objects that they encapsulate against unauthorized access. However, the security model should also define and classify the forms that attacks may take, providing a basis for the analysis of threats to the system. In the model protection is described in terms of objects, but the concepts apply equally well to resources of all types.

Protecting objects Figure 48 (page 104) shows a server that manages a collection of objects on behalf of some users. The users run a client program that can send invocations to the server to perform operations on the objects. The server carries out the operation specified in each invocation and sends the result to the client. Objects are intended to be used in different ways by different users. For example, some objects may hold a user's private data, while other objects may hold shared data. To support this *access rights* specify who is allowed to perform the operations on an objects (e.g. who is able to read or write its state). This is done by including users in our model as the beneficiaries of access rights. To each invocation and each result the authority on which it is issued is associated. Such an authority is called a *principal*. A principal may be a user or a process. The server is responsible for verifying the identity of the principal behind each invocation and checking that the principal has sufficient access rights to perform the requested operation on the particular object invoked, rejecting requests that do not. The client may check the identity of the principal behind the server to ensure that the result comes from the required server.

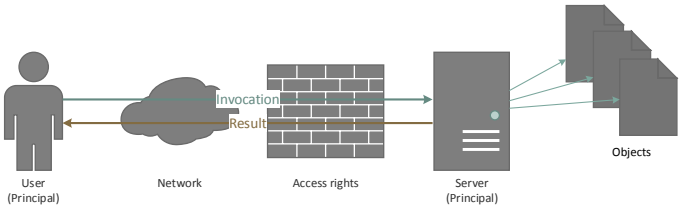


Figure 48: Basic model showing a server managing a collection of objects protected by access rights for a user. The users connects over a network to access the server.

Securing processes and their interactions Processes interact by sending messages over a network/communication service. These messages are exposed to attack because the network and the communication service that they use are open,

to enable any pair of processes to interact. Servers and peer processes expose their interfaces, enabling invocations to be sent to them by any other process. Distributed systems are often deployed and used in tasks that are likely to be subject to external attacks by hostile users, especially if the system is designed to handle financial transactions, confidential or classified information or any other information whose secrecy and integrity is crucial. Integrity is threatened by security violations as well as communication failures.

The enemy To model security threats, we need to define an enemy (or adversary) that is capable of sending any message to any process and reading or copying any message sent between a pair of processes. Such attacks can be made simply by using a computer connected to a network to run a program that reads network messages addressed to other computers on the network, or a program that generates messages that make false requests to services, purporting to come from authorized users. The attack may come from a computer that is legitimately connected to the network or from one that is connected in an authorized manner. This is shown in Figure 49 (page 105).

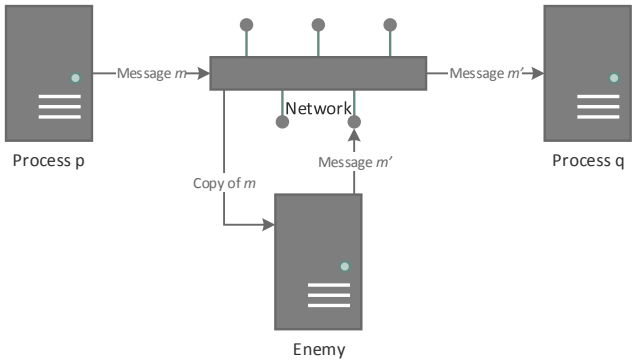


Figure 49: An enemy intercepting the message m from process p to process q, replacing message m with the modified message m'.

The threats from a potential enemy include *threats to processes* and *threats to communication channels*.

Threats to processes A process that is designed to handle incoming requests may receive a message from any other processes in the distributed system, and it cannot necessarily determine the identity of the sender. While communication protocols such as the **I**nternet **P**rotocol (IP) include the address of the source computer in each message, it is not difficult for an enemy to generate a message with a forged source address [44]. This lack of reliable knowledge of the source of a message is a

threat to the correct functioning of both servers and clients.

Since servers can receive invocations from many different clients, it cannot necessarily determine the identity of the principal behind any particular invocation. Even if a server requires the inclusion of the principal's identity in each invocation, an enemy might generate an invocation with a false identity. Without reliable knowledge of the sender's identity, a server cannot tell whether to perform an operation or reject it.

For clients the situation is not very different, when the client receives the result of an invocation from a server, it cannot necessarily tell whether the source of the result message is from the intended server or from an enemy, which might be 'spoofing' the server. Thus the client could receive a result that was unrelated to the original invocation.

Threats to communication channels An enemy can copy, alter or inject messages as they travel across the network and its intervening gateways. Such attacks present at threat to the privacy and integrity of information as it travels over the network and to the integrity of the system.

Another form of attack is the attempt to save copies of messages and to replay them at a later time, making it possible to reuse the same message over and over again. All these threats can be defeated by the use of *secure channels*, which we will describe below.

Secure channels The use of a secure channel allows a pair of processes to communicate in a way such that each of the processes knows reliably the identity of the principal on whose behalf the other process is executing. Therefore if a client and server communicate via a secure channel, the server knows the identity of the principal behind the invocations and can check their access rights before performing an operation. This enables the server to protect its objects correctly and allows the client to be sure that it is receiving results from a valid server. Also, a secure channel should ensure the privacy and integrity (protection against tampering) of the data transmitted across it. Finally, each message should include a physical or logical timestamp (such as a sequence number) to prevent messages from being replayed or reordered.

A1.2.3 Failure model

In a distributed system both processes and communication channels may fail, thus departing from what is considered to be correct or desirable behaviour. The failure model defines the ways in which a failure may occur in order to provide an understanding of the effects of failures. In [42] they distinguish between failures of processes and failures of communication channels. These failures can be categorized as *omission failures*, *arbitrary failures* and *timing failures*. A graphical representation of the severity of the failures is given in Figure 50 (page 107). A description of each failure is given in this section.

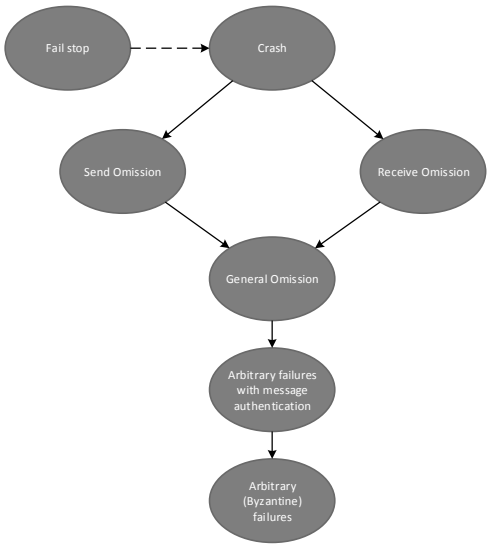


Figure 50: Shows the possible failures to both processes and communication channels. The severity of the failures increases downwards, thus a fail-stop is less severe than a byzantine failure.

Omission failures Consider the communication between two processes as in Figure 51 (page 108). Process p wants to send a message m over the network to process q. To do so, it places the message in its outgoing message buffer. The communication channel transports the message over the network and places it into the incoming message buffer of process q. Process q receives the message by taking m out of the incoming message buffer.

The communication channel produces an omission failure if it does not transport a message from p's outgoing message buffer to q's incoming message buffer. This so-called 'message dropping' is generally caused by lack of buffer space at the receiver or by a network transmission error. In [42] a loss of messages between the sending process and the outgoing message buffer is known as a *send omission failure*, and

a loss of messages between the incoming message buffer and the receiving process is known as a *receive omission failure*. Loss of messages in between is known as a *channel omission failure*.

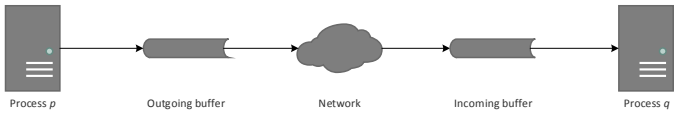


Figure 51: Process p places a message m in the outgoing buffer, which is then taken and transported over the network the the incoming buffer of process q and eventually delivered to the process q.

Besides communication channels, processes can also produce omission failures. The most important omission failure of a process is a *crash*. When a process has crashed we mean that it is halted and will not execute any further steps of its program, ever. The design of services that can survive in the presence of faults can be simplified if it can be assumed that the services on which the depend crash cleanly, that is, their processes either function correctly or else stop. A process crash is called a *fail-stop* if other processes can detect certainly that the process has crashed. For example, in a synchronous system a process crash can be detected by using timeouts. If the process does not respond within the timeout, it is assumed to have crashed. In an asynchronous system a process could broadcast a message that it is crashing before actually halting its operation.

Arbitrary failures While omission failures are not convenient, they are benign failures. The worst set of failure semantics are described as arbitrary or *Byzantine* failures, in which any type of error may occur. For example, a process may set wrong values in its data items, or may return a wrong value in response to an invocation. An arbitrary failure of a process is one in which it arbitrarily omits intended processing steps or takes unintended processing steps. Arbitrary failures can not be detected by seeing whether the process responds to invocations, because it might arbitrarily omit to reply.

Communication channels can also suffer from arbitrary failures, message contents may be corrupted, nonexistent messages may be delivered or real messages may be delivered more than once, for example. Arbitrary failures of communication channels are rare[37] however, since communication software is able to recognize them and reject the faulty messages.

Timing failures As you might expect, timing failures are only applicable to synchronous distributed systems because there is a limit set on the process execution

time, message delivery time and clock drift rate. From this follow the failures listed in Table 10. In an asynchronous distributed system, an overloaded server may respond too slowly, but we cannot say that it has a timing failure since there is no guarantee on the response time.

Class of failure	Affects	Description
Clock	Process	The process's local clock exceeds the bounds on its drift rate from real time
Performance	Process	Process exceeds the bounds on the interval between two steps.
Performance	Channel	A message's transmission takes longer than the stated bound.

Table 10: Overview of timing failures

Masking failures from clients Each component in a distributed system is generally constructed from a collection of other components. It is possible to construct reliable services from components that exhibit failures. For example, multiple servers that hold replicas of data can continue to provide a service when one of them crashes. A knowledge of the failure characteristics of a component can enable a new service to be designed to mask the failure of the components on which it depends. A service can mask a failure by either hiding it completely, or by converting it into a more acceptable failure. For example, using a checksum to mask a corrupted message, converting a arbitrary failure into an omission failure.

A basic communication channel can exhibit the omission failures described earlier, however it is possible to build a communication service that masks some of those failure. *Reliable communication* is defined in terms of validity (any message in the outgoing message buffer is eventually delivered to the incoming message buffer) and integrity (the message received is identical to the one sent, and no message is delivered twice).

The obvious threats to integrity come from two sources: first, any protocol that retransmits messages but does not reject a message that arrives twice. And second, malicious users that may inject spurious messages, replay old messages or tamper with messages. The first threat can be countered by attaching a sequence number to messages, so the protocol can detect when a message is delivered twice. And the second threat can be countered by using the secure channels described earlier.

A2 ACID Properties

The ACID properties are a set of properties that guarantee that database *transactions* are processed reliably [45]. A transaction is a single logical operation on the data. This means that a series of statements, such as creating an order and updating the amount of stock, are conceptually whole as a single transaction. ACID is an acronym for ***A**tomicity*, ***C**onsistency*, ***I**solation* and ***D**urability*.

A2.1 Atomicity

Atomicity requires that each transaction is 'all or nothing', if any part of the transaction fails, the entire transaction will fail. This leaves the database in a state as if the transaction never happened. Atomicity should be guaranteed in all situations, including power failures, errors, and crashes.

A2.2 Consistency

The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules. Note that this consistency guarantee only considers the rules defined in the database, thus in case of a developer error or bug in the application, data might be inconsistent from the application's point of view.

A2.3 Isolation

The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if the transactions were executed serially, or one after another.

A2.4 Durability

Durability ensures that once a transaction has been committed, the changes are permanent. Even in the event of a power loss, crash, or error immediately after completing the transaction.

A3 SOLID principles

SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion) is a mnemonic acronym introduced by Michael Feathers for the "first five principles" identified by Robert C. Martin in the early 2000s that stands for five basic principles of object-oriented programming and design. The principles when applied together intend to make it more likely that a programmer will create a system that is easy to maintain and extend over time.

This chapter is largely copied from Wikipedia.

A3.1 Single responsibility principle

The single responsibility principle states that every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.

The term was introduced by Robert C. Martin in an article by the same name as part of his Principles of Object Oriented Design [46], made popular by his book Agile Software Development, Principles, Patterns, and Practices [47]. Martin described it as being based on the principle of cohesion, as described by Tom DeMarco in his book Structured Analysis and Systems Specification [48].

Martin defines a responsibility as a reason to change, and concludes that a class or module should have one, and only one, reason to change. As an example, consider a module that compiles and prints a report. Such a module can be changed for two reasons. First, the content of the report can change. Second, the format of the report can change. These two things change for very different causes; one substantive, and one cosmetic. The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times. The reason it is important to keep a class focused on a single concern is that it makes the class more robust.

Continuing with the foregoing example, if there is a change to the report compilation process, there is greater danger that the printing code will break if it is part of the same class.

A3.2 Open-closed principle

The open/closed principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification" [49] that is, such an entity can allow its behaviour to be modified without altering its source code. This is especially valuable in a production environment, where changes to source code may necessitate code reviews, unit tests, and other such procedures to qualify it for use in a product: code obeying the principle doesn't change when it is extended, and therefore needs no such effort.

A3.3 Liskov substitution principle

Substitutability is a principle in object-oriented programming. It states that, in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may be substituted for objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.).

More formally, the Liskov substitution principle (LSP) is a particular definition of a subtyping relation, called (strong) behavioral subtyping, that was initially introduced by Barbara Liskov in a 1987 conference keynote address entitled Data abstraction and hierarchy. It is a semantic rather than merely syntactic relation because it intends to guarantee semantic interoperability of types in a hierarchy, object types in particular. Barbara Liskov and Jeannette Wing formulated the principle succinctly in a 1994 paper as follows: *Let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be provable for objects y of type S where S is a subtype of T.*

A3.4 Interface segregation principle

The interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use [47]. ISP splits interfaces which are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called role interfaces. ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy.

A3.5 Dependency inversion principle

the dependency inversion principle refers to a specific form of decoupling where conventional dependency relationships established from high-level, policy-setting mod-

ules to low-level, dependency modules are inverted (i.e. reversed) for the purpose of rendering high-level modules independent of the low-level module implementation details. The principle states:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions. The principle inverts the way some people may think about object-oriented design, dictating that both high- and low-level objects must depend on the same abstraction.

A4 Design Patterns

A Design Pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations.

Patterns are formalized best practices that the programmer must implement themselves in the application. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

This chapter is largely copied from Wikipedia.

A4.1 Abstract Factory

The Abstract Factory pattern is a software creational design pattern first described in [50]. It provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes [50]. In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic interface of the factory to create the concrete objects that are part of the theme. The client does not know (or care) which concrete objects it gets from each of these internal factories, since it uses only the generic interfaces of their products.

This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface.

The structure of the Abstract Factory pattern is shown in Figure 52 (page 115).

A4.2 Strategy

The Strategy pattern (also known as the policy pattern) is a software design pattern, whereby an algorithm's behaviour can be selected at runtime. Formally speaking, the strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it. Strategy is one of the patterns included in [50] that popularized the concept of using patterns in software design.

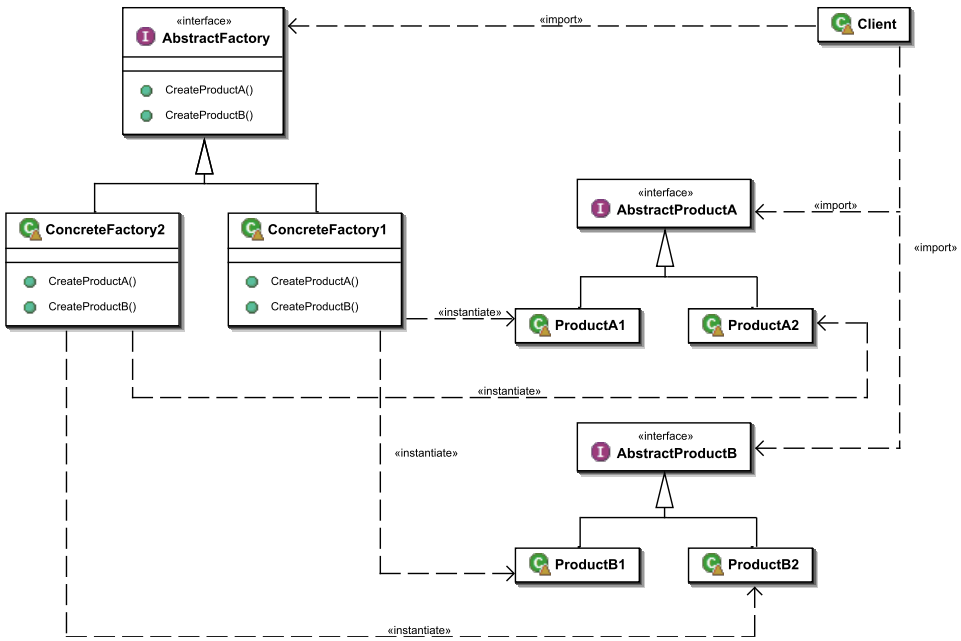


Figure 52: Structure of the Abstract Factory Pattern

For instance, a class that performs validation on incoming data may use a strategy pattern to select a validation algorithm based on the type of data, the source of the data, user choice, and/or other discriminating factors. These factors are not known for each case until run-time, and may require radically different validation to be performed. The validation strategies, encapsulated separately from the validating object, may be used by other validating objects in different areas of the system (or even different systems) without code duplication.

A4.3 Object Pool

The object pool pattern is a software creational design pattern that uses a set of initialized objects kept ready to use, rather than allocating and destroying them on demand. A client of the pool will request an object from the pool and perform operations on the returned object. When the client has finished, it returns the object, which is a specific type of factory object, to the pool rather than destroying it.

Object pooling can offer a significant performance boost in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instances in use at any one time is low. The pooled object is obtained in predictable time when creation of the new objects (especially over network) may

take variable time.

A4.4 Singleton

The singleton pattern is a design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects. The term comes from the mathematical concept of a singleton.