The Sea of Stuff: a Model to Manage Shared Mutable Data in a Distributed Environment

Simone Ivan Conte

University of
St Andrews

This thesis is submitted in partial fulfilment for the degree of

Doctor of Philosophy (PhD)

at the University of St Andrews

August 2018

Abstract

Managing data is one of the main challenges in distributed systems and computer science in general. Data is created, shared, and managed across heterogeneous distributed systems of users, services, applications, and devices without a clear and comprehensive data model. This technological fragmentation and lack of a common data model result in a poor understanding of what data is, how it evolves over time, how it should be managed in a distributed system, and how it should be protected and shared. From a user perspective, for example, backing up data over multiple devices is a hard and error-prone process, or synchronising data with a cloud storage service can result in conflicts and unpredictable behaviours.

This thesis identifies three challenges in data management: (1) how to extend the current data abstractions so that content, for example, is accessible irrespective of its location, versionable, and easy to distribute; (2) how to enable transparent data storage relative to locations, users, applications, and services; and (3) how to allow data owners to protect data against malicious users and automatically control content over a distributed system. These challenges are studied in detail in relation to the current state of the art and addressed throughout the rest of the thesis.

The artefact of this work is the Sea of Stuff (SOS), a generic data model of immutable self-describing location-independent entities that allow the construction of a distributed system where data is accessible and organised irrespective of its location, easy to protect, and can be automatically managed according to a set of user-defined rules.

The evaluation of this thesis demonstrates the viability of the SOS model for managing data in a distributed system and using user-defined rules to automatically manage data across multiple nodes.

The code for this work can be found online at the following URL:
`https://github.com/sea-of-stuff` (GNU GPL v3).

Declaration

Candidate's Declaration

I, Simone Ivan Conte, do hereby certify that this thesis, submitted for the degree of PhD, which is approximately 66,400 words in length, has been written by me, and that it is the record of work carried out by me, or principally by myself in collaboration with others as acknowledged, and that it has not been submitted in any previous application for any degree.

I was admitted as a research student at the University of St Andrews in September 2014.

I received funding from an organisation or institution and have acknowledged the funder(s) in the full text of my thesis.

Date Signature of candidate

Supervisor's Declaration

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of PhD in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date Signature of supervisor

Permission for Electronic Publication

In submitting this thesis to the University of St Andrews we understand that we are giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. We also understand, unless exempt by an award of an embargo as requested below, that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that this thesis will be electronically accessible for personal or research use and that the library has the right to migrate this thesis into new electronic forms as required to ensure continued access to the thesis.

I, Simone Ivan Conte, confirm that my thesis does not contain any third-party material that requires copyright clearance.

The following is an agreed request by candidate and supervisor regarding the publication of this thesis:

- No embargo on print copy.
- No embargo on electronic copy.

Date Signature of candidate

Date Signature of supervisor

Underpinning Research Data or Digital Outputs

Candidate's declaration

I, Simone Ivan Conte, hereby certify that no requirements to deposit original research data or digital outputs apply to this thesis and that, where appropriate, secondary data used have been referenced in the full text of my thesis.

Date Signature of candidate

Acknowledgments

I would like to thank my supervisors, Alan Dearle and Graham Kirby, for their guidance and endless patience throughout my doctoral studies.

Thanks goes to Adrian O'Lenskie and Ian Paterson, from Adobe Systems Inc., for their precious advice and support.

I would like to acknowledge the School of Computer Science, which has made my stay in St Andrews, over the last eight years, enjoyable academically and has given me the opportunity to meet people that have been a true inspiration to me. Thanks goes to Stuart Norcross and the *Fixit* team who have helped me with the provision and management of the testbed used for the experiments.

Thanks to my office mates Masih, Tom, Ward, and Ryo who have provided me with challenging, interesting, and fun discussions on a daily basis.

I would like to thank my aunt Giusy for inspiring me, when I was ten, to pursue a career in science.

I am forever grateful to my parents and my brother for their immense and invaluable love and support, which helped me arrive where I am today.

Finally, the biggest thanks goes to Giulia, who makes me smile everyday.

Funding

This work was supported by *Adobe Systems, Inc.* and *EPSRC* [grant number `EP/M506631/1`].

Contents

Abstract . . . iii

Declaration . . . v

Permission for Electronic Publication . . . vi

Underpinning Research Data or Digital Outputs . . . vii

Acknowledgments . . . ix

1 Introduction 1

1.1 Introduction . . . 1

1.2 The Three Challenges . . . 2

1.2.1 Limitation of the Current Data Storage Abstractions . . . 3

1.2.2 Transparent Data Storage . . . 5

1.2.3 Data Ownership, Protection and Control . . . 7

1.3 Hypothesis . . . 8

1.4 Thesis Contributions . . . 10

1.5 Thesis Structure . . . 11

2 Background 13

2.1 Data Storage Concepts . . . 13

2.1.1 Data . . . 13

2.1.2 Location and Naming . . . 14

2.1.3 Metadata . . . 17

2.1.4 Caching . . . 19

2.1.5 CAP Theorem . . . 20

2.1.6 Replication, Erasure Coding, and Resiliency . . . 20

2.1.7 Scalability 24

2.1.8 Security 25

2.2 Data Management Systems 32

2.2.1 File Systems 33

2.2.2 Database Systems 41

2.2.3 Versioning in Storage Systems 46

2.2.4 Networked File Systems 49

2.2.5 Cloud Storage 54

2.2.6 Object Storage 55

3 Literature Review **57**

3.1 File Systems 57

3.1.1 Extended Attributes Support 58

3.1.2 Tagged Files 59

3.2 Networked File Systems 59

3.2.1 The Hadoop File System and Google File System 60

3.2.2 GlusterFS 63

3.3 Versioning in Storage Systems 64

3.3.1 Manual Data Versioning 64

3.3.2 Versioning in Backup Applications 65

3.3.3 Version Control Systems 67

3.4 Cloud Storage 77

3.4.1 Infrastructure as a Service Storage 77

3.4.2 Software as a Service Storage 80

3.4.3 Multi-Cloud Storage 86

3.5 P2P 87

3.5.1 Overlay Networks 87

3.5.2 P2P Storage Systems 88

3.6 Context-Aware Storage 99

3.6.1 The Semantic File System 101

3.6.2 The quFile 101

3.7 Conclusions 102

4 Design Requirements **105**

4.1 End-User Requirements 105

4.2 Model Requirements 106

4.3 Architecture Requirements 107

5 The Sea of Stuff **109**

5.1 The Sea of Stuff Model 109

5.1.1 The Sea of Stuff GUID 109

5.1.2 A Manifest-based Model 110

5.1.3 The Node Model 111

5.1.4 The Data Model 112

5.1.5 The Metadata Model 127

5.1.6 The User-Role Model 130

5.1.7 The Context Model 138

5.1.8 Summary of the Sea of Stuff Model 152

5.2 The Sea of Stuff Architecture 153

5.2.1 The Node 154

5.2.2 Services 155

5.2.3 Considerations 166

6 Reference Implementation **167**

6.1 Services 167

6.1.1 Storage Management Service 167

6.1.2 Manifest-Data Management Service 171

6.1.3 Metadata Management Service 173

6.1.4 Node Management Service 173

6.1.5 User-Role Management Service 173
6.1.6 Context Management Service 173
6.1.7 Node Maintenance 174
6.1.8 REST API 174
6.1.9 Instrumentation 174
6.2 Node Storage 176
6.3 Node Configuration 178
6.4 Example Applications 178
6.4.1 WebApp 178
6.4.2 SOS-WebDAV Server 180
6.5 Overall Prototype Limitations 182

7 Comparative Evaluation 183

7.1 Evaluation of Requirements 184
7.1.1 End-User Requirements 184
7.1.2 Model Requirements 185
7.1.3 Architecture Requirements 186
7.2 File Systems 188
7.2.1 The File Metaphor 188
7.2.2 The Directory Metaphor 189
7.3 Networked File Systems 190
7.3.1 Client-Server File Systems 190
7.3.2 Clustered File Systems 191
7.3.3 Shared-Nothing File Systems 191
7.4 Versioning in Storage Systems 192
7.4.1 Backup Applications 192
7.4.2 Version Control Systems 194
7.5 Cloud Storage 200
7.5.1 Infrastructure as a Service Storage 200

7.5.2 Software as a Service Storage 204

7.5.3 Multi-Cloud Storage 207

7.6 P2P 207

7.6.1 OceanStore and Pond 207

7.6.2 InterPlanetary File System 208

7.6.3 Perkeep 211

7.6.4 Tahoe-LAFS 212

7.7 Context-Aware Storage 213

7.7.1 The Semantic File System 214

7.7.2 The quFile 215

7.8 Conclusions 216

8 Experimental Evaluation 219

8.1 Experiments Outline 219

8.2 Experimental Methodology 221

8.3 Experimental Setup 221

8.4 The Experimental Framework 223

8.4.1 Extension of the Sea of Stuff Node 223

8.4.2 The Experimental Framework Utility 223

8.4.3 The Datasets 225

8.5 Reference Implementation Benchmarks 226

8.5.1 Input/Output Benchmark 226

8.5.2 Network Latency and Throughput 230

8.5.3 Atoms and Manifests Replication 231

8.6 Context Experiments 232

8.6.1 The Nature of the Predicates 233

8.6.2 Predicates and the Domain 238

8.6.3 The Nature of the Policies 243

8.6.4 Policies and the Codomain 245

8.6.5 Contexts under Failure . . . 247
8.7 Conclusions . . . 260

9 Conclusions and Future Work 263
9.1 Thesis Summary . . . 263
9.2 Review of Contributions . . . 264
9.3 Limitations of Evaluations . . . 265
9.4 Future Work . . . 266
9.4.1 The Sea of Stuff Model and Architecture . . . 266
9.4.2 Security . . . 267
9.4.3 Sea of Stuff Integration with Existing Technology . . . 268
9.5 Concluding Remarks . . . 268

Bibliography 269

Image Credits 295

List of Abbreviations 297

Glossary of Terms 301

Appendices 303

A Settings 305
A.1 SOS Node Settings . . . 305

B Experiment Settings 309
B.1 Butts Wynd Cluster . . . 309
B.2 Experiment Configuration . . . 310

C Libraries used by the SOS prototype 313
C.1 Castore . . . 313
C.2 guid-sta . . . 313

C.3 St Andrews Utilities 314
C.4 Apache Tika 314
C.5 Unirest 314
C.6 WebDAV-server 314
C.7 fs-sta 314

D Systems Comparison 315

Introduction

1.1 Introduction

Among historians, history is defined as "the study of past events" and is commonly known to have started when humans have invented writing [1]. Any event prior to the invention of writing, instead, is known to have happened in the prehistoric era. With the invention of writing, humans have started to record facts, events, trading exchanges, stories, thoughts, and anything else we could possibly imagine. With the ability to write, humans started not only to produce information, but also to store it: inside caves, on stones, papyri, wood, and, later, paper. Humans did not just produce and store information, but they did so in ever growing amounts. Throughout history, the invention of papyrus, paper, and the printing press has allowed information to be stored and replicated more easily, faster and at a lower cost than before. However, none of these technologies has had the same impact on information storage as the invention of the digital computer.

The two main building stones of computing are: data and algorithms [2]. Data and algorithms are so important in computing that sometimes computer science itself is defined as the science of processing data [3, 4, 5, 6]. In this work we are going to focus mainly on data and the algorithms used to manage data. Looking at the history of computing from a data-centric perspective, it is interesting to note how the digital computing revolution has allowed us to generate and store data at an ever-growing pace. It has been estimated that the size of the "digital universe" has grown by approximately 16 zettabytes[1] in 2016

[1] One zettabyte (1 ZB) = 10^{21}bytes = 1 trillion gigabytes.

only and the trend is exponential [7, 8]. Together with the internet, we are now able not only to create and accumulate data, but also to do so while sharing it across the globe. The computing revolution has not slowed down ever since the invention of the internet, with the number of devices and amount of data constantly increasing. The result of the increase of connectivity is data flowing in all directions and users constantly sharing photos, videos, documents, *etc.* In addition, most people own multiple computing devices, which often synchronise data with cloud services and send it to other devices and users. However, how the internet and its web services work is much more complex than sending and receiving data from a user's device to another. For example, a cloud-based backup application replicates data to multiple locations hidden to the user and synchronises it to the user's devices, but the actual behaviour is hard to predict and understand. The model controlling the data, in fact, is complex and bound to the specific service. Given this complexity, it is necessary to study how data management can be improved and simplified, and create processes and algorithms built around data concepts that are not bound to a specific service, application, or user, and that are easy to understand for the user, easy to implement, efficient to execute, and complete enough to capture the needs of today and adapted to the needs of tomorrow.

To summarise, even though data has become pervasive and the difficulty in using digital devices has decreased drastically, data management is still challenging. This thesis identifies three challenges that users face: (1) how to extend the current data abstractions, which have been around for decades but have not adapted to the needs of today's applications and users, so that content, for example, is versionable, easy to distribute, and accessible irrespective of its location; (2) how to enable transparent data storage relative to locations, applications, and services; and (3) how to allow data owners to properly and effectively protect data and automatically control it over a distributed system.

1.2 The Three Challenges

The *Background* and *Literature Review* chapters discuss the problems highlighted by these three challenges in more depth and breadth.

1.2.1 Limitation of the Current Data Storage Abstractions

The family of persistent data storage systems can be logically subdivided in two main branches: file systems and databases. As the name suggests, file systems are designed and built around the concept of the file; whereas databases are designed around the idea of storing data, possibly related to each other, in an organised and efficient manner. File systems are also commonly associated with the hierarchical structure of files and directories, a concept any computer user is familiar and comfortable using. Hierarchical file systems, however, have been around for almost half-century and have not evolved rapidly enough to cope with the user needs and the new technologies [9, 10, 11, 12].

In [13], Jim Gray states that "everything builds from files as a base", from composite data formats to databases (*i.e.*, tables are usually stored as files), but some file systems have actually slowly become database systems themselves.[2] Gray is not suggesting that all file systems have become databases, but rather that the file system abstraction is outdated and must be re-thought. Take, for instance, the growing trend for end-users to use application-centric systems over data-centric systems. Application-centric systems, like Android[3], iOS[4], or the Windows 8's metro[5], tend to hide the hierarchical structure of the underlying file system while enhancing it with the help of a better user-interface and other services. Application-centric systems also hide the file metaphor as much as possible and simply present data via applications. An application, for example, may use a metadata database to provide additional information about the data displayed or it can interact with an external data storage service for better data reliability and data sharing. These are only two of many examples of how applications and services are being created around data-centric systems, such as file systems, to facilitate complex actions, such as sharing or data replication, and to hide complexities, such as a complex hierarchical file system structure. However, actual storage system implementations are not forced to fall within one category: file systems or databases. In fact, it is quite common for a storage

[2] It should be noted that some databases run on raw partitions and not file systems.

[3] Android. `https://android.com/` [last accessed on 11/10/2017].

[4] iOS. `https://apple.com/ios/` [last accessed on 11/10/2017].

[5] Microsoft Windows. `https://microsoft.com/windows/` [last accessed on 11/10/2017].

system to have traits from both branches, as suggested in [13].

When new software is released, users can benefit not only from new features, but also from new 'mental models' about data. As a result, users interact with data in new ways and both the users and the systems acquire new behaviours. One of the side effects of this is a misalignment and lack of correspondence between the 'real model' and the 'mental model' of how files are understood [14]. The most basic data operations: **c**reate, **r**ead, **u**pdate, **d**elete (CRUD) [15] are well understood by most users and it is easy to associate these operations with files when working locally. The standard file metaphor, however, is overloaded and much more complex and hard to define [16, 17]. In fact, the number of operations on files is larger than the simple CRUD operations. Users want to move, rename, duplicate, share, synchronise, protect, associate, tag, replicate, version, sign files.[6] Each of these operations makes the original file definition, based on CRUD, more complex and harder to understand. Let's take, for example, the 'synchronise' operation. When taking a photo with a smartphone, this is saved to the local storage of the device and a backup application (or applications) will start synchronising it to an external service, which itself will replicate the photo to multiple data servers transparently to the user. Naïvely, users understand the 'synchronise' operation as a simple 'copy' operation, even though there are more underlying processes happening, which can also lead to behaviours difficult to predict. For example, a synchronisation process might involve two nodes that act on the same data, stored locally, independently of each other so that conflicts may arise. In this scenario, one node attempts to propagate a data change while the other a data deletion. The resulting state of the system, whether the data is deleted on the first node or copied back on the second node, does not always reflect the user's intentions.

In particular, this thesis claims that storage abstractions should address the following:

- **Data abstraction over locations**. Data should be accessible irrespective of where it is stored and from where it is accessed.

[6] This is not a comprehensive list of all the operations that are allowed on files, and data in general, that users wish to accomplish.

- **Versioning**. It should be possible to version data, at arbitrary granularities, so that users can see how data has changed and revert to previous changes if needed. Versioning models also favour collaboration over data.

- **Rich metadata support**. Metadata is used to improve retrieval, usage, and management of data. Therefore, richer metadata results in better data storage services.

Data storage abstractions should also be designed and built taking into account the following non-functional requirements:

- **Implementation independence**. The abstractions should be described as concepts independent from their actual implementation.

- **Scalability**. These abstractions should be usable in a system that scales globally.

- **Data integrity**. Users and/or services operating on such abstractions should be able to verify whether a data/metadata has changed or not.

- **Security**. Users and/or services should be able to protect data/metadata from being accessed from unwanted parties.

1.2.2 Transparent Data Storage

The aim of a data storage solution providing transparent properties is to hide any complexities from its users. Transparency can be applied over many aspects of a system, but often only location transparency is addressed. Location transparency ensures that the name associated with the data is related to the actual data rather than to its locations. This work, however, suggests that data storage systems should also address transparency over applications and services.

A feature that makes cloud storage applications and services very useful, and popular, is their ability to abstract the stored content from locations. Whether data is stored in a cloud storage application, like Dropbox [18] or OneDrive [19], or in a cloud storage service, such as Amazon S3 [20] or Rackspace Cloud Storage [21], the location of the data is transparent to users or applications. Users do not have to remember all the locations of the different replicas of the stored data. As a benefit, the data stored will comply with the service-level agreement of the service, meaning that data is stored with high-reliability, -availability and -durability at a reasonable, and often little, cost. Relying on cloud storage, means also less maintenance, relying on the work of hundreds of thousands of engineers hours, and good scalability. Storing data in the 'cloud', however, can also present some issues:

- Services can go down, even if they promise 99.9999+%[7] availability/reliability/durability.
- Data is locked-in: users have little or no control over the data stored in the cloud.[8]
- The company providing the service can go bankrupt.
- Data might not be stored safely and securely [22, 23, 24, 25].

Moreover, even though cloud services abstract data over their locations, data is still not abstracted over location-services. For example, imagine having a version control repository in your local file system, which is also synchronised with a remote version control service and backed-up to a cloud storage service. The file system, the remote version control service, and the cloud storage service each provide location abstraction, but each of them also represents a location-service, which is not abstracted to the user.

A common solution to have transparency over cloud services is to build a '*cloud of clouds*' (CoC), where a generic data management solution provides abstraction over multiple cloud services [26, 27, 28, 29, 30]. CoC, however, have the following limitations:

[7] For a time-based property, a six-nine probability (99.9999%) is equivalent to 2.6 seconds downtime per month, or 31.2 seconds downtime per year.

[8] For example, most cloud storage solutions allow users and/or applications to choose the geographical regions where data is stored. However, while advantageous, this level of abstraction is still limiting, as one is often not able to describe precisely where data should be stored. Plus, it is usually the case that extending this level of control over multiple cloud solutions is not possible.
In cloud storage, data deletion is also an issue, since one cannot delete data directly and must trust that the cloud service is properly deleting it.

- Their generic abstraction provides a smaller subset of functionalities, so as to adapt to many cloud services (or it provides many features, but not all of them for all services).

- Moving content across the underlying cloud services is still challenging.

- Moving content while preserving its provenance, metadata and integrity is still hard.

- Data is still locked-in, but in multiple cloud services rather than one.

1.2.3 Data Ownership, Protection and Control

The third challenge is related to how to allow users to explicitly state ownership over data, how to effectively protect data and how to automatically control it over a distributed system. These three sub-challenges are linked together as they question what users can do with their own data.

1.2.3.1 Ownership

Data ownership is the relationship between a user and some arbitrary data that he/she 'possesses'. In digital systems data ownership is implemented by mapping the concept of user to data. However, the concept of user, as well as the type of mapping of the data, is strictly related to the system in question. In most systems, the concepts of users and ownership are not built in the data model of the system, but are rather built on top of it. Having the concepts of users and ownerships separated from the data model results in a non-uniform behaviour and management when the system is distributed and/or has to interact with other systems.

1.2.3.2 Protection

Related to ownership is protection. Protection is the ability to stop unwanted users to freely access data. With data sharing being central to distributed storage, protection is a fundamental aspect to provide a proper service. In most traditional storage systems, for example, data is protected by simply assigning password protected users to it (*i.e.*, authentication), but the actual data is not protected via encryption. Cloud storage services, instead, protect users' data via encryption, but users have no control over the actual protection mechanism since this is not part of the data model, but part of the cloud service architecture.

1.2.3.3 Control

Finally, the third sub-challenge is related to how to allow users to control their data in a distributed environment and how to let the system perform control automatically. This works defines *control over data* as the ability to decide and act upon the data, such as copying, encrypting, or deleting it. Controlling data in a distributed system is challenging and even more so when the task has to be run autonomously. The current commercial solutions that allow control over content are *ad-hoc* tools, such as Syncthing[9] and Resilio Sync[10]. Large distributed storage systems and cloud storage services adopt automatic data control mechanisms, but they require specialised technical knowledge that the typical end-user does not have.

1.3 Hypothesis

The ability to correctly and easily store and control data is hard to achieve. Ideally, one would wish to have a system that satisfies the following properties:

1. Storage space is infinite.
2. The system minimises the latency of access to data.
3. Data is always available regardless of its location and the locations of users.
4. The system abstracts the physical location of data and metadata.
5. Access to data and metadata can be controlled.
6. Data and metadata may be shared between globally distributed users.
7. Arbitrary levels of resilience may be defined and automatically enforced.

[9] Syncthing. `https://syncthing.net/` [last accessed on 03/11/2017].

[10] Resilio. `https://resilio.com/` [last accessed on 03/11/2017].

8. All data and metadata may be versioned.

The above properties are hard to achieve due to theoretical and physical limits. This thesis attempts to approximate to these properties at two distinct levels: (1) at a level that is visible to users and (2) at the system level — as realised by a computational model.

This thesis hypothesises that:

H1:

> *A data model based on immutable, self-describing, re-computable, and content-addressable entities can be used to build a distributed data management system which approximates to the properties 1–2 and satisfy the properties 3–8 above in a manner that provides advantages over existing storage systems.*

H2:

> *A data model based on immutable, self-describing, re-computable, and content-addressable entities can be used to build a personal distributed data management system where data is automatically managed based on user-defined rules in a manner which provides advantages over existing solutions and approaches.*

The distributed storage model and the prototype developed in this work attempt to provide the following properties:

1. **Location abstraction**. Users can perform data operations irrespective of where the data is stored and from where it is accessed.
2. **Copying, Moving and Sharing**. Data and metadata can be copied, moved, and shared across locations, users, and services at different granularities while retaining their properties.

3. **Data Protection**. Data and metadata can be protected independently of locations and of each other. The result is the ability to protect data once and replicate it many times. It is also possible to protect data, while leaving metadata readable to everyone — if one wishes to increase the data's discoverability — without necessarily giving access to the actual content.

4. **Versioning**. Content can be versioned at different granularities.

5. **Autonomic Data Management**. Users can automatically organise and enforce actions on data using well-defined rules.

6. **Resiliency**. Resiliency over data is supported by the ability of the model to abstract entities over their locations. Entities are not trapped by a single system and can be replicated across multiple nodes or services without any loss of information, thus increasing the level of availability in the face of faults.

7. **Users and Roles**. Users can play multiple roles and access different data views based on the role played.

8. **Absence of Central Authority**. Entities can be exchanged across individual nodes without the need for a central authority.

To test the hypotheses **H1** and **H2**, a proposed data model was designed, prototyped, and later evaluated (see Chapter 5 for the design of the SOS and Chapters 7 and 8 for its evaluation).

1.4 Thesis Contributions

The contributions of this work are:

1. The design of a generic model — the **Sea of Stuff** — for managing data in a heterogeneous distributed system.

2. The design of a distributed architecture for managing data.

3. A comprehensive literature review on data management.

4. An implementation of the model and architecture proposed.

5. An evaluation of the Sea of Stuff model.

6. The identification of a set of future research directions.

1.5 Thesis Structure

This chapter — **Introduction** — introduced the problems, challenges, and the research hypothesis of this thesis and listed the contributions of this work.

Chapter 2 — **Background** — presents a summary of data management concepts necessary to read the rest of the thesis. This chapter also provides an overview of different storage systems and their properties.

Chapter 3 — **Literature Review** — presents the state of the art in data management along with the dimensions identified by the hypotheses, as outlined in this chapter.

Chapter 4 — **Design Requirements** — outlines the needed requirements from the end-user perspective, as well as the requirements need to design and implement the proposed model.

Chapter 5 — **The Sea of Stuff** — presents the design of the Sea of Stuff model, reasoning about the design decisions made along the way and how it compares to the related work. Furthermore, this chapter briefly described the prototype developed with its features and limitations.

Chapter 7 — **Comparative Evaluation** — evaluates the Sea of Stuff model qualitatively against the solutions presented in the background and literature review chapters.

Chapter 8 — **Experimental Evaluation** — evaluates the Sea of Stuff model quantitatively through experimentations on the prototype developed.

Chapter 9 — **Conclusions and Future Work** — concludes this thesis and presents a summary of future work directions.

Background

This chapter introduces the basic concepts about data storage systems and provides an introduction to the different classes of storage solutions.

Data storage is the study of *recording* and *reading* data into storage media (*e.g.*, volatile memory, hard disk, recording tape), and consists of (a) designing and implementing the actual physical storage media and (b) designing and developing the software to allow the operating system and/or the applications to use the storage media. This thesis will ignore (a), as those are tasks and challenges belonging to the study of electronics, and will focus on how storage is abstracted and managed via software. In particular, this thesis will focus on the high-level techniques[11], systems, and solutions to store persistent data locally and in a distributed setting.

2.1 Data Storage Concepts

2.1.1 Data

Data[12], in informatics, is defined as a sequence of bits — which can be created, read, processed, saved, and stored digitally — and constitutes the basic building block for any data storage system.

The way data is organised and stored — such as a file, a database entry, or a composition of independent data elements — affects also the data '*metaphor*' perceived by users and the set of operations associated with it (*e.g.*, create, read, update, delete, share, copy, move,

[11] Techniques and methods on how to read, write, and process data on physical disks and manage IO in general are not the focus of this work and will not be taken into account.

[12] The word data is the plural of the Latin word *datum* which stands for *thing given*.

etc.). The most relevant data abstractions are discussed in more detail in the relevant sections about the different types of data storage systems (see Section 2.2) and in the *Literature Review* chapter.

2.1.2 Location and Naming

2.1.2.1 Location

Location is the "the place" where data is stored and from which it can be fetched. However, this definition is ambiguous, since it is not always clear what and where this place is and what the relationship with the logical data is. So we must distinguish location into two types: the physical location and the logical location of the data.

The **physical location** of data indicates the place where it is stored on physical storage media, such as a disk or a recording tape. However, understanding how data is managed in different physical locations is beyond the scope of this work and will not be discussed further in this thesis.

The **logical location** of data is the place where it is stored in relation to other data or other abstractions, such as directories for file systems, from a user/application level perspective. In a flat file system, which has only the root directory, the logical location of the files and their names are one and the same. In hierarchical file systems, the logical location of a file is related to its name and the structure of directories that contain it. Directories are the abstractions that allow files to be organised in hierarchies. The textual representation of logical locations is the **path**. The following, for example, is a path for a file contained under the *desktop* directory, which itself is contained under the *home* directory: `home/desktop/a_file`

The path of a file consists of the definition of the name of the file and the description of the logical location of the file in relation to directories and other files. Paths can be **absolute** or **relative**. Absolute paths locate a file regardless of the current working directory the user is in, while a relative path — as the name suggests — is a path that is built relatively to another path.

2.1.2.2 Naming

"Naming is a mapping between logical and physical objects" [31, p. 707] and allows users to refer to files and directories using human-readable names even though these are physically stored in a storage media and accessed by using physical "hard-to-read" addresses.[13] The resolution of the naming mappings, such as the one below, is performed by the **name service** of a system. A typical name service for a local file system resolves the user-level name to its physical address, which corresponds to its actual location, as following:

user level textual name → *system level identifier* → *physical address*

Naming is also very important in distributed systems, where data is distributed across multiple nodes (see *Glossary of Terms* for the definition of node). In distributed systems, naming and location are related one another and the following two properties must be taken into account when looking at naming in a system [31, 32]:

- **Location Transparency**. The name associated with the data is not related to its physical storage location.
- **Location Independence**. The name associated with the data does not need to be changed when its physical storage location changes. A name service providing location independence dynamically maps names to locations.

Finally, the collection of all the names allowed in a system defines the **namespace**. There are two types of namespaces: **local** and **global**. Names in a local namespace are unique only within a given node or set of machines. Systems using a local namespace can concatenate the host name with the data name to provide a wider access to such data. The web is an example of a system where file names are concatenated to a host name (*i.e.*, http://⟨domain⟩/⟨filename⟩) in the form of a URI (Uniform Resource Identifier). This technique, however, conflicts with the location transparency and location independence

[13] A physical address is a sequence of digits that indicate the actual location of the data on the physical storage and it is hard-to-read for humans.

properties described above, since the URI is related to the location of the data and needs to be changed if the data is moved somewhere else. On the other hand, names within a global namespace are unique over all the nodes of a system.

The remainder of this section briefly describes the URI, the globally unique identifier (GUID), and content-addressable naming schemes, necessary to understand the next chapters.

Uniform Resource Identifier

A URI is a string of characters that identifies a resource and follows the general scheme:

```
protocol:[[user[ password]@]host[ port]][path][?query][#fragment]
```

which is described in detail in the RFCs 2396 [33] and 3986 [34].

There are two main types of URIs: **locator identifiers**, or **URL** (Uniform Resource Locator), and **name identifiers**, or **URN** (Uniform Resource Name). URLs define the location of the resources, with the *protocol* describing how resources should be accessed. Examples of URLs are:

- http://www.ietf.org/rfc/rfc2396.txt
- mailto:foo@bar.com
- https://doi.org/10.1109/JPROC.2010.2096170

URNs, unlike URLs, do not contain any hint about the location of the content. Examples of URNs are:

- urn:ietf:rfc:2648
- urn:guid:7A7B95D784007B930599D491366E0C272D119EB0
- urn:isbn:0451450523

Globally Unique Identifier

A GUID (Globally Unique Identifier) is a number, of any length, used to uniquely identify a resource in a system. That means that two resources cannot have the same GUID, unless they are actually representing the same entity (*i.e.*, they are the same sequence of bytes). The RFC 4122 [35] defines a GUID as a 128-bit number that can be generated in multiple ways (*e.g.*, using a hash function) and that describes a resource uniquely within the system adopting it. GUIDs shorter or longer than 128-bit can also exist. The following is an example of GUID in hexadecimal format (32 digits, 4 bits each):

```
123e4567-e89b-12d3-a456-426655440000
```

Content-Addressable Naming Schemes

A content-addressable naming scheme allows storage systems to bind data to names that are related to its actual sequence of bytes, such that a different sequence of bytes must be bound to another name. In such schemes, the name of some content is usually generated by applying a cryptographically secure hash function on the content (see Section 2.1.8.2), so that a change to the content results also in a change in the name.

The purpose of content-addressable naming schemes is to abstract data from locations, so that the users of such storage system are able to access data irrespective of where it is stored or where it is accessed from.

2.1.3 Metadata

Metadata is "data about data" and it is used to facilitate retrieval, usage, and management of content. Metadata is usually structured data defined by well known standards [36, 37, 38], but it can take any arbitrary form. Jenn Riley defines metadata as "key to the functionality of the systems holding the content, enabling users to find items of interest, record essential information about them, and share that information with others" [38]. Windows- and Unix-based file systems, for example, support a limited set of metadata attributes (*e.g.*, name, creation/modification/last access date, owner) that are used to search, order, and understand the stored data. Application-based data storage systems — such as Google Photo[14], Amazon Kindle[15], or social networks like Facebook[16] or Twitter[17] — support richer metadata linkage with data. This, however, usually comes at the cost of binding the data within the application ecosystem.

Metadata can also be generated by a system to enable certain features. For example, distributed storage systems, such as the Google File System and GlusterFS, use *location metadata* to map data with its storage locations. Likewise, desktop search applications, such as Spotlight[18] and Alfred[19], index and enrich local data to provide better search functions (*e.g.*, find all the files authored by the user Simone, all the pdf files opened last week, or all the images tagged as *holiday*).

For the purpose of this work, metadata is classified based on the relationship it has with the data it describes:

Intrinsic metadata

Intrinsic metadata is metadata inherent to the data and embedded within it. We distinguish two types of intrinsic metadata: *explicit* and *implicit*. Explicit intrinsic metadata (EIM) is structured metadata stored within the data, and thus easy to extract using metadata extractors. Examples of EIM are EXIF for JPEG or XMP for JPEG, PSB and PDF. Implicit intrinsic metadata (IIM), on the other hand, is metadata that can be algorithmically extracted from the data. IIM is computationally expensive to retrieve. Examples of IIM are: whether there is an animal in a JPEG photo, if the photo is mostly blue, the tone of the voice in an audio or the speech-to-text translation of an audio file.

[14] Google Photo. `https://google.com/photos/about/` [last accessed on 15/08/2018].

[15] Amazon Kindle. `https://amazon.co.uk/kindle-dbs/fd/kcp` [last accessed on 15/08/2018].

[16] Facebook. `https://facebook.com` [last accessed on 15/08/2018].

[17] Twitter. `https://twitter.com` [last accessed on 15/08/2018].

[18] "Spotlight is a fast desktop search technology that allows users to organise and search for files based on metadata". Spotlight is provided by Apple and available for Apple's macOS and iOS operating systems. `https://developer.apple.com/library/content/documentation/Carbon/Conceptual/MetadataIntro/MetadataIntro.html` [last accessed on 09/10/2017].

[19] Alfred. `https://alfredapp.com/` [last accessed on 09/10/2017].

Extrinsic metadata

Extrinsic metadata is external to the data to which it refers and it may be stored *explicitly* in a metadata-like manifest or provided *implicitly* by some underlying storage system. Explicit extrinsic metadata (EEM) is metadata provided by applications and/or users. Examples of EEM are tags or metadata stored within the pathnames.[20] Implicit extrinsic metadata (IEM), instead, is metadata provided by the native system, such as the creation/modification/last-access times or the size of the data.

2.1.4 Caching

Caching is the practice of storing content temporarily in a fast storage location to improve IO performance. Specifically, the operation of caching consists of storing data to a location with faster access. Data might be cached for various reasons: due to an actor of the system, or external to it, that requests some data; due to temporal locality of the data; or because of the spatial locality of data.

In a local storage system, data is cached in the RAM (random-access memory) or in the L1-2-3 CPU internal caches. In a distributed system caching occurs also at the node level. Specifically, data can be cached in four places: the server's disk, the server's memory, the client's memory, and the client's disk. Irrespective of the number and types of nodes of a system, the individual interactions among them are always going to involve two nodes, one of which acts as a client and the other as a server. The purpose of caching in a distributed system is to store data to a node that is closer, geographically or topologically, to the client so that latency is reduced.

Upon retrieving data, the cache is queried first. If the data is found, it is said that a **cache hit** occurred and the data is returned. Otherwise, a **cache miss** occurred and the data is retrieved from another, slower, location (*e.g.*, the local disk or another node). Depending on the strategy in place, the retrieved data is added to the cache so that future requests will result in cache hits.

When caching data, one should take into account the following three issues: (1) the

[20] For example, information about the content loaded under the path *home/photo/holiday/* can be inferred by the naming of the directories.

cached data may not be coherent[21] across all cache locations; (2) maintaining the cache coherent and up-to-date[22] can become expensive and cumbersome over time; (3) the chosen coherency model may not always satisfy everyone's needs.

2.1.5 CAP Theorem

The CAP theorem, formulated by Eric Brewer, states that in a distributed system it is possible to achieve only two of the following three properties:

- **C**onsistency. The data is consistent across all available nodes.
- **A**vailability. The requested data is always available at all reachable nodes.
- Tolerance to Network **P**artitions. The system is tolerant to failure of parts of the network [39, 40].

The CAP theorem acts as a "rule of thumb" for distributed system designers and it is commonly assumed that any distributed system must forfeit one of the three properties. However, Brewer later rectified that the "2 of 3" rule is misleading, since some trade-off of the three properties is also achievable [41]. In fact, the CAP theorem defines the above constraints only when a system is under failure or not running under optimal conditions. To the support of a more flexible CAP theorem, Abadi has proposed the PACELC[23] formulation, which poses the following question: "if there is a partition (P), how does the system trade off availability and consistency (A and C); else (E), when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C)?" [42].

2.1.6 Replication, Erasure Coding, and Resiliency

2.1.6.1 Replication

Data replication is the process of copying — replicating — data to one or more locations in order to achieve better data availability and stronger resistance to faults. For instance, if

[21] Coherency means that all clients requesting the cached data, should get the same exact data.

[22] The latest changes to the data.

[23] PACELC is pronounced as "pass-elk".

one node crashes or one piece of data is corrupted, the data can still be recovered from the other replicas. Replication is also important to improve performance of data management in a distributed system that requires scaling, quantitatively and geographically. Typically, replication is used to balance load between multiple nodes [32, 43]. Additionally, in a distributed system, data replication can be used to achieve caching by locating data over strategic locations.

Replicating some data D by a replication factor n, results in an overall space cost of n and a system that can tolerate $n-1$ replica failures. A technique to increase resilience is to divide D in smaller blocks, which can be distributed independently. In the first approach, D is stored in n locations; while in the second approach the blocks can be stored over $n \times c$ locations, where $c = (size(D)/size(block))$. This means that the number of locations that need to be unavailable for D to become also unavailable is higher in the latter case. The downside of the latter, however, is that managing blocks that are replicated over many more locations can be expensive. Chunking data into blocks also result in a system with better throughput, since blocks can be transferred concurrently.

Overall, when integrating data replication in a system, the following should be taken into account:

- Replicating data to a remote location has a cost which depends on multiple factors related to such location: network properties, quality of service, and the actual price of buying, renting, and/or maintaining the storage location.

- Where to replicate data. A system might want to replicate data closer to where it is accessed, but also spread it across multiple distant geographical locations, so that storage is more resistant to geographical related issues.

- How many replicas to have for some given data (the replication factor). Increasing the replication factor increases also the redundancy, availability, and fault resistance of the data, but it also requires more storage space, which has a cost in terms of management, money, energy, and actual physical space.

- When to replicate data. A system might optimise its replication process by replicating data only when necessary, for example when a particular piece of data is more requested.

- Keeping track of where the replicas are has a cost, since the storage nodes storing the replicas should be periodically queried.

- Keeping the replicas consistent has also a cost, since it might involve complicated algorithms such as the two-phase commit [44] or the paxos [45] protocols.

- Some of the replicas might become out-of-date and policies to update or discard them should be designed.

Data Consistency

When considering consistency for data storage in distributed systems, we look at two main approaches: *eventual consistency* and *read-after-write consistency* (sometimes known as *strong consistency*). Eventual consistency is achieved if all the replica locations eventually contain the same data, without any constraints on when consistency will be achieved. However, in a system with high churn[24], eventual consistency may never be achieved. In the read-after-write consistency model, instead, data is consistently available to be read immediately after it has been written. A system implementing a read-after-write consistency model usually forfeits availability in favour of consistency (see CAP Theorem, 2.1.5).

2.1.6.2 Erasure Coding

An alternative approach to achieve high redundancy, availability, and resiliency while saving storage space is via erasure coding [46, 47, 48, 49]. Erasure coding is an error-correction technique that extends some data D consisting of n symbols with some additional k symbols, such that only n symbols are needed to re-construct the original data. A symbol could be a single bit or byte of data, or, more commonly, a block of data. Erasure coding is

[24] Churn is a measure of the number of nodes that join or leave a given system.

used to manage volumes of disks, for some RAID levels (Redundant Array of Independent Disks) [50], or in distributed systems [51], by distributing $n+k$ blocks over multiple nodes.

The general erasure coding mechanism consists of dividing some data D in n data blocks and calculating k *coding* blocks, such that the storage space cost is $n+k$ and the system can tolerate k block failures. The advantage of erasure coding over simple replication (where data is chunked into smaller blocks), is that it is possible to achieve the same level of fault tolerance, but with a lower storage cost, as outlined by the ratio:

$$\frac{storage\ cost(erasure\ coding)}{storage\ cost(replication)} = \frac{c+k}{c \times (k+1)};$$

where:

$c =$ is the number of blocks that the data is divided into

$k =$ is the number of block failures that can be tolerated

It follows, for example, that choosing (n, k) to be equal to $(10, 4)$ for erasure coding and $(10, 2)$ for simple data replication (3 replicas), results in a 53.4% storage space saving and the ability to tolerate 4 block failures instead of 2.

However, maintaining the level of resilience constant when parts of a system fail can be expensive. The naïve approach for recovering a data block requires the reconstruction of the data with n blocks to recalculate the missing one. This puts a significant stress on the system network bandwidth and overall disk IO. More efficient repair mechanisms have been designed, but require more complex coordination among the components of the system [52, 53, 54]. Different block placement strategies can also have a significant impact on the system performance [55].

A technique to increase erasure coding write performance is to divide the data D in blocks and apply erasure coding for each block, so that when updating D, erasure coding has to be reprocessed only for the blocks that have changed. This approach is used by Tahoe-LAFS [56] (see *Literature Review* Chapter, Section 3.5.2.4) and MaidSAFE [57].

2.1.6.3 Resiliency

Resiliency, or fault tolerance, is the ability of a system to continue to provide its service in the face of faults (*e.g.*, a part of the system does not return the expected result, a remote server does not respond or produced an error message, the network is partitioned). Resiliency is a very important property in distributed systems, where failures are the norm, not the exception. For a system to be classified as resilient, two requirements must be satisfied:

1. The system can continue to provide a service even if some of its components are down or malfunctioning.
2. Components of the system can automatically recover from failures without affecting the behaviour and performance of other components.

2.1.7 Scalability

Scalability is the ability of a system to cope with growth in the amount of work to be done. A system is said to be scalable even if its behaviour degrades definitely and noticeably, as long as such degradation remains acceptable to the users of the system. In the case of storage systems, scalability is generally considered along the following axes:

- The number of data entities.
- The size of data entities.
- How data is organised (*e.g.*, through directories for files, or buckets for objects in cloud object storage services).
- The number of nodes, in a networked or distributed storage system, that can be supported.
- The number of clients that can be served by a node simultaneously.
- The amount of requests that a node can satisfy.

Scalability can be addressed at the software level, by designing and building a system that can handle more workload within the same time frame, and at the hardware level, by either scaling horizontally (adding more nodes) or vertically (adding more resources to the existing nodes of the system). Horizontal scaling is a common approach since it can be achieved using commodity hardware, but it requires more complex coordination among all machines. Vertical scaling, instead, is easier to implement, but more expensive and limited by the amount of resources that can be added to a single machine.

2.1.8 Security

Computer security is the field that studies how to protect a computer system from being corrupted or damaged, information from being read without authorization, and a service from being disrupted. Security is a vast topic, so this section will only cover the concepts relevant to the understanding of this work.

2.1.8.1 Ownership, Access Control and Capabilities

One of the basic protection mechanisms in security is the ability to assign ownership over data, define who has access to it, and what operations a given user or a collection of users, a group, is allowed to perform. This section illustrates the basic protection mechanisms employed by Unix file systems.

The owner/group/other Unix permission model

The owner/group/other Unix permission model is the most common mechanism to control access over files and directories. In this model, each file or directory is associated with its **owner** (*e.g.*, sic2, al, gnck), a named **group** (*e.g.*, students, staff), and **other**, a special group used to indicate any user of the system. Users and groups are associated with unique system identifiers, ‘uid’ and ‘gid’ respectively. The example in Snippet 2.1 shows the owner/group/other permissions of some files through the use of the *ls* utility.

The owner/group/other Unix permission model, however, provides only limited access control for systems that have more than one user and/or one group. This limitation is overcome using access control lists or capabilities (see below).

```
$ ls -l
-rw-r--r--  1 sic2 students    314 30 Aug 14:40 README.md
-rw-r--r--  1 al   staff     24093 25 Nov 17:37 bibliography.bib
drwxr-xr-x 12 sic2 students    384 28 Nov 11:04 chapters
lrwxr-xr-x  6 gnck staff       192 27 Nov 17:50 experiments -> /Users/sic2/git/experiments
-rw-r--r--@ 1 gnck staff     25349 27 Nov 17:59 experiments.xlsx
-rw-r--r--  1 sic2 students   6444 27 Nov 17:07 main.tex
```

Snippet 2.1: Example of ‘ls -l’ output showing the permissions on files and directories. The first column indicates the mode of the listed entity. The mode is composed by the file type (*e.g.*, ‘-’ for a regular file, ‘d’ for directory, ‘l’ for symbolic link), and three permission classes for user, group, and other. Each permission class consists of the triplet ‘rwx’ (read, write, execute). The third column indicates the user-owner, while the fourth column indicates the group-owner.

Access Control Lists

Access control list (ACL) is a security mechanism to enforce more fine-grained control over who can access some given data and what operations a user, or group, is allowed to perform [58]. In ACL, a file or a directory is associated with a list of users/groups and the type of access they have been granted (*e.g.*, read, write, execute, *etc.*). Snippet 2.2 illustrates an example of ACL in a Unix file system. Windows systems use a similar approach by using a list of access control entries [59].

ACLs enable complex access control patterns, however, its maintenance can be challenging, especially in a system with many users and/or groups.

```
$ getfacl -d main.tex
# file: main.tex
# owner: sic2
# group: students
user::rwx              # Base ACL Entry, mask not applied
user:gnck:rwx          # Effective:r-x
user:al:rwx            # Effective:r-x
group::rwx             # Base ACL Entry, effective:r-x
group:staff:r-x
```

```
mask:r-x
other:r-x             # Base ACL Entry, , mask not applied
# The default ACL can be specified for files only
default:user::rwx
default:user:gnck:rwx # Effective:r-x
default:group::r-x
default:mask:r-x
default:other:---
```

Snippet 2.2: Example of an ACL list.

Capabilities

Capabilities are a security technique that defines what operations a user is allowed to perform on some given data. A capability is an unforgeable token, or key, that grants its owner permissions to access an entity, such as a file, and to perform operations upon it [60, 61, 62, 63]. Further, a capability could also be the address of the entity, as suggested by Fabry [60], so that the address is independent of the location of the entity and a change in location of the entity does not involve any change in the capability.

Similarly to ACL, capabilities can also be hard to maintain, such as the process of revoking access to a particular user to a particular entity, when this user has already made a copy of the entity.

2.1.8.2 Data Integrity

Data integrity is the property of data to be consistent over time. Data integrity is an important property in the context of security and a system that supports data integrity should be able to detect unintentional data changes or data integrity loss, such as when data changes due to hardware faults or a malicious user changes the data.

One of the techniques used to verify data integrity is to generate hashes from the data. One such hash function is **MD5**. The MD5 algorithm is a hash function generating 128-bit hash values (see RFC 1321 [64]). The algorithm, however, has been proven broken multiple times over the years, making it vulnerable to malicious attackers [65, 66]. More popular and secure alternatives to the MD5 hash function are the **SHA** cryptographic hash functions. The SHA algorithms are a family of hash functions created and published under the National Institute of Standards and Technology (NIST). As of the time of writing, the SHA family comprises four sub-family of algorithms: SHA-0, SHA-1, SHA-2, and SHA-3. SHA-0 was dismissed after a few years of its publication and so it is rarely used [67, 68]. The SHA-1 hash function produces a 160-bit hash value, usually represented as a hexadecimal string 40-character long. An example of the result of a SHA-1 function is the following:

```
SHA-1(‘‘The quick brown fox jumps over the lazy dog’’) =
2fd4e1c67a2d28fced849ee1bb76e7391b93eb12
```

SHA-1, unlike the SHA-0, has been more resistant to collisions[25], which were found only recently in early 2017 [69]. The SHA-1 is widely used across a large number of applications and systems. In storage systems, the SHA-1 function is used to uniquely identify content.[26] The advantage of using a cryptographic function is that (1) it outputs a fixed size sequence of bits; (2) it is infeasible to reverse; and (3) it can be used to verify the integrity of the data that generated the hash value. Thus, if one bit of the data has changed, then re-calculating the hash function will generate a new non-matching hash value. Some version control systems use SHA-1 to hash both the versioned data as well as a reference to the previous versions of the data (*i.e.*, the history of the repository). Using this approach, it is possible to verify the integrity of the versioned content as well as of the history of the repository (see Sections 2.2.3 and 3.3 to know more about version control systems). The SHA-2 and SHA-3 algorithms produce larger hash-values and provide better security against collisions. However, SHA-2 and SHA-3 have not seen a wide use until recently, as both computation power and storage space have improved and become cheaper.

```
SHA-2-256(‘‘The quick brown fox jumps over the lazy dog’’) =
d7a8fbb307d7809469ca9abcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592
```

[25] A collision is the event when a function maps two different inputs to the same output.

[26] SHA-1 is not collision free, but the probability of generating a random pair of hashes that collides is less than one in 2^{80}.

```
SHA-3-256(‘‘The quick brown fox jumps over the lazy dog’’) =

69070dda01975c8c120c3aada1b282394e7f032fa9cf32f4cb2259a0897dfc04
```

Algorithm (and variant)	Output size (bits)	Max. message size	Security bits	Published year
MD5	128	Unlimited	<64	1992
SHA-0	160	2^{64} - 1	<34	1993
SHA-1	160	2^{64} - 1	<63	1995
SHA-2 (-256)	256	2^{64} - 1	128	2001
SHA-2 (-512)	512	2^{128} - 1	256	2001
SHA-3 (-256)	256	Unlimited	128	2015
SHA-3 (-512)	512	Unlimited	256	2015

Table 2.1: The most relevant cryptographic hash functions. The number of security bits is a measure of the number of possible outputs a hash function has and therefore indicates how hard it is to find an input with a given output.

2.1.8.3 Data Protection via Encryption

In addition to ensuring data integrity, sometimes one may wish to protect data from being read by unwanted parties. In order to protect data, encryption algorithms are adopted. Encryption algorithms allow bytes of data, that we want to protect, to be transformed to another sequence of bytes using an encryption key, so that it becomes very hard to transform such encrypted data back to its original form, unless the decryption key is known. The encryption algorithms can be classified as **symmetric** or **asymmetric**. Without going into the mathematics of encryption algorithms, symmetric-key encryption requires a single key to both encrypt and decrypt the data; while asymmetric encryption requires a pair of keys, where one is used to encrypt the data and the other one to decrypt it. A common use of asymmetric encryption is one where one of the keys (public key) can be distributed publicly without compromising the security enforced by the other matching key (private key). Asymmetric encryption is generally more expensive than symmetric encryption, so its usage has often been limited to encryption of small data, key management by encrypting symmetric keys, and digital signature [70]. The diagram in Figure 2.1 illustrates the difference between symmetric-key and asymmetric-key encryption.

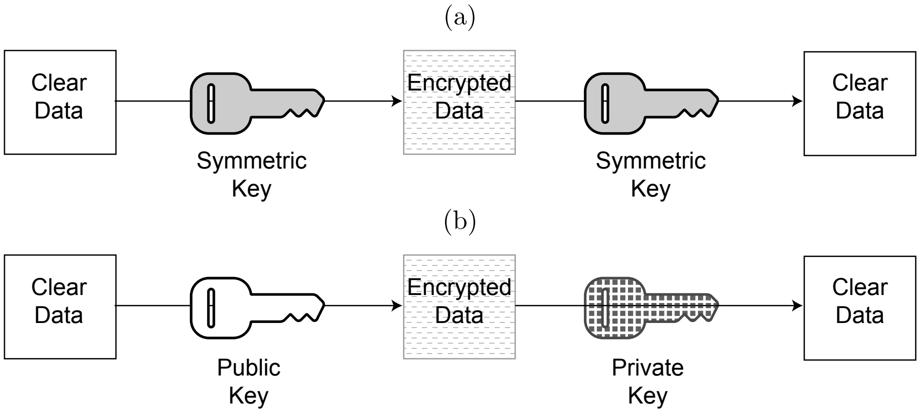


Figure 2.1: A schematic of (a) symmetric encryption and (b) asymmetric encryption. In (a) the symmetric key is used to both encrypt and decrypt the data. In (b) the public key is used to encrypt the data, while the private key is used to decrypt it.

Two relevant symmetric-key algorithms are: DES and AES. The Data Encryption Standard (DES) was developed by IBM in the '70s and has been considered the standard symmetric-key algorithm for about two decades [71]. The Advanced Encryption Standard (AES) algorithm uses a block size of 128 bits and employs keys of three different lengths: 128, 196, and 256 bits. AES was published in 1998 and it has now taken over DES [69].

One of the most used asymmetric algorithms is RSA (Rivest-Shamir-Adleman), which is based on the creation of a public key given two large prime numbers that should be kept secret and whose product is very hard to factor [72]. The RSA algorithm is slow, therefore it is usually used to encrypt shared symmetric-keys rather than for encrypting the data itself.

2.1.8.4 Digital Signatures

A digital signature is a type of cryptographic algorithm used to ensure authentication, non-repudiation and integrity over some data [73, 74]. Authentication allows to determine whether the originator of the data is really whom he/she claims to be; non-repudiation prevents the signer to deny that he/she was the one signing the content; and integrity, as previously explained in Section 2.1.8.2, verifies that the data has not changed since the signature has been generated. The digital signature protocol consists of three steps:

- Key-generation of a private and public key-pair.
- Signing the content using the private key, resulting in a signature.
- Verification of the signature using the public key.

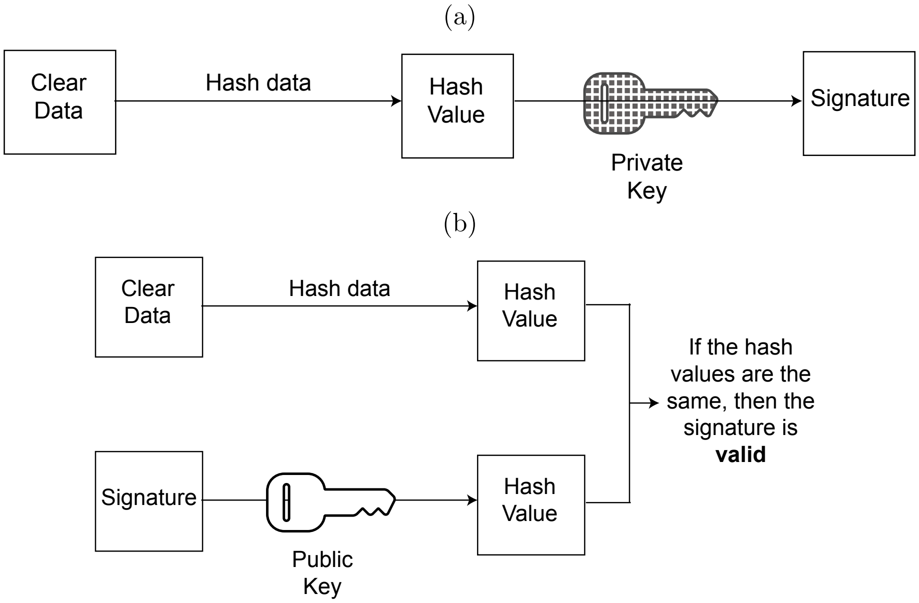


Figure 2.2: A schematic of how digital signatures work. (a) shows the signing process for some given data. First the data is hashed, using a cryptographic hash algorithm, and then the resulting hash value is encrypted using the private key. The result is the signature for the clear data. (b) shows the steps for verifying a given signature. The signature is decrypted using the public key of the signer, then the result is compared with the expected hash value of the data. If the two hash values are the same, then the signature is considered valid.

It is good practice to digitally sign the hash of the data rather than the data itself (see the schematics in Figure 2.2). Signing the hash of the data is faster than using an asymmetric encryption algorithm directly on the data. The RSA algorithm, for instance, can only handle a limited amount of data. In addition, hashing the data first results in a shorter digital signature, which is then easier to distribute. The Digital Signature Algorithm (DSA) is an implementation of the Digital Signature Standard and the most widely used digital signature algorithm [75].

2.2 Data Management Systems

Data management and storage systems vary for their intrinsic properties, their architecture and what they are used for. The high number of types and implementations of data management systems (DMS) limits the amount of them that can be described in this thesis as well as the level of detail that can be covered. This section examines the following classes of DMS, which have been selected based on the challenges described in the *Introduction* Chapter and the system proposed in the *The Sea of Stuff* Chapter:

- File Systems
- Database Systems
- Version Control Systems
- Networked File Systems
- Cloud Storage Systems
- Object Storage Systems

For each of these classes of systems only the relevant level of detail is covered. For example, database systems are described in terms of their data abstractions, rather than their internal implementations. Similarly, networked file systems are described in terms of their abstractions and interactions between their components, without any description of the actual protocols involved. Thus, the remaining of this chapter should be considered as an introductory reading to storage systems. A similar approach has been taken for the *Literature Review* Chapter. In order to keep the chapter concise, some storage implementations belonging to the classes below have not been referenced or described and some classes of DMS have not been included, such as content management systems, digital management systems, or remote object models (*e.g.*, CORBA[27]).

[27] Corba. `http://www.corba.org/` [last visited on 25/11/2018)

2.2.1 File Systems

The operating system (OS) is a software program that manages the computer hardware as well as the software resources for the user applications. The majority of OS provide an overlapping set of services: process management and coordination, memory management, IO and storage management. The file system is a subsystem of the operating system that provides data management over secondary storage.[28]

2.2.1.1 Files and Directories

Files and directories are the two most important file system user-level abstractions, or metaphors.

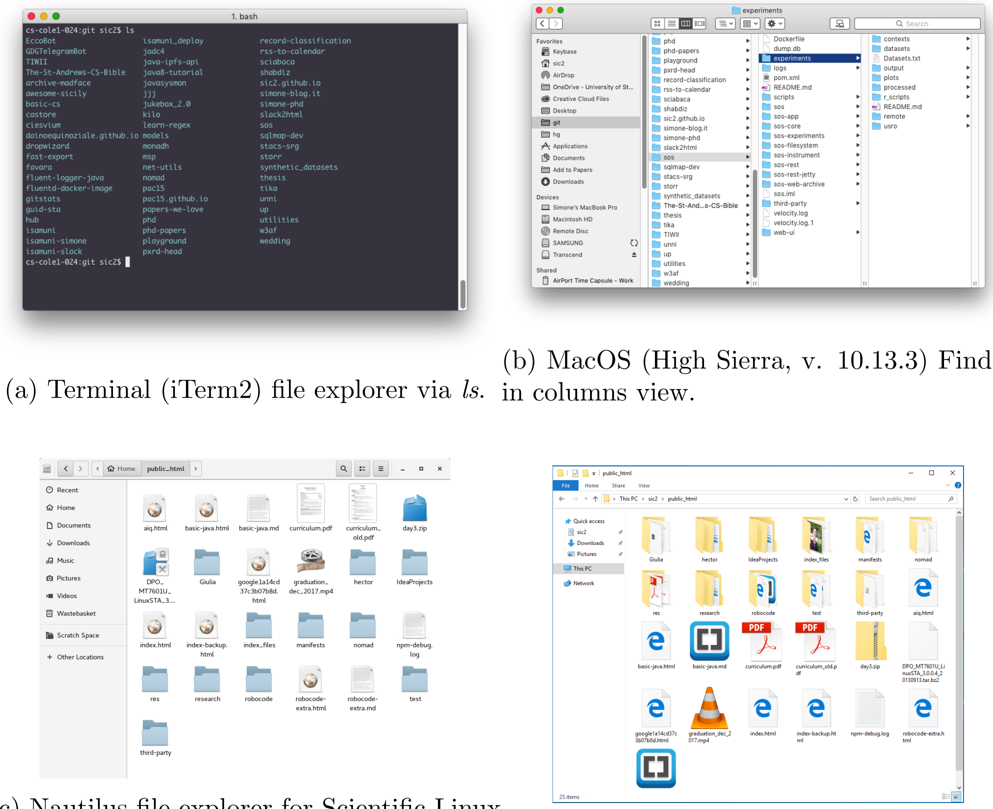
Traditionally, the file is defined as “a collection of related information defined by its creator [...]” and “files represent programs (both source and object forms) and data” [31, p. 26]. Alternatively, the definition of a file corresponds to the definition of data, so that a file is simply a sequence of bits stored on some storage media. This second definition, however, can be misleading as it merges together the concept of file with the one of data, and it does not include the metadata information related to the file as well as the operations that can be carried out over a file.

The user-level metaphor of a file consists of data, attributes, and operations. The attributes define the metadata information used by the OS to describe the file and to access the actual data. The two main attributes of a file are the name and the identifier. The name is a human-readable string, known also as user-level textual name, which is of use only to users and/or applications. The identifier, instead, is a unique identifier, within the file system, for the file and is also known as *system-level identifier.* Section 2.1.2 discusses in more details the role of naming in storage systems, in particular the relationship between naming and location.

The other attributes of a file are, but are not limited to: type, location[29], size, pro-

[28] Secondary storage is non-volatile memory, not directly accessible by the CPU, which usually stores large amount of data compared to primary storage, at the cost of slower IO.

[29] The location attribute contains information about the storage device where the file is stored and the location within that device.



```
cs-cole1-024:git sic2$ ls
EccoBot                      isamuni_deploy              record-classification
GDGTelegramBot               jadc4                       rss-to-calendar
TIWII                        java-ipfs-api               sciabaca
The-St-Andrews-CS-Bible      java8-tutorial              shabdiz
archive-madface              javasysmon                  sic2.github.io
awesome-sicily               jjj                         simone-blog.it
basic-cs                     jukebox_2.0                 simone-phd
castore                      kilo                        slack2html
ciesvium                     learn-regex                 sos
dainoequinoziale.github.io models                      sqlmap-dev
dropwizard                   monadh                      stacs-srg
fast-export                  msp                         storr
favara                       net-utils                   synthetic_datasets
fluent-logger-java           nomad                       thesis
fluentd-docker-image         pac15                       tika
gitstats                     pac15.github.io             unni
guid-sta                     papers-we-love              up
hub                          phd                         utilities
isamuni                      phd-papers                  w3af
isamuni-simone               playground                  wedding
isamuni-slack                pxrd-head
cs-cole1-024:git sic2$ ▌
```

(a) Terminal (iTerm2) file explorer via *ls*.

(b) MacOS (High Sierra, v. 10.13.3) Finder in columns view.

(c) Nautilus file explorer for Scientific Linux (v. 7.4 with GNOME 3.22.2) in icons view.

(d) Windows 10 file explorer in icons view.

Figure 2.3: Example of some typical file explorers for different operating systems.

tection, and statistical information (*e.g.*, time of last access or time of last modification). These attributes, such as the location, size and protection are necessary for the operating system to actually access the data and perform various file operations. The most important file operations are: file creation, data writing and reading of a file, data search (seek), file deletion, and file truncation.

The other user-level abstraction, common to many file systems, is the *directory* or *folder*, which is used to impose a logical organisational structure for files and other directories in a file system. A Unix-like directory is simply a file containing a table of symbols that translates human-readable file names to the corresponding file identifiers or other directory identifiers. Figure 2.3 shows some examples of files and directories as seen by users via file managers (or file browsers) or a terminal window.

2.2.1.2 File System Types

File systems can be classified in different ways. From a user perspective, files systems can be classified by how files and directories are organised. From an implementation perspective one might look at how the file system is actually implemented.

The most common organisation for files and directories, from a user-perspective, is the hierarchical one (see Figure 2.4). In a hierarchical file system, entities are linked through a path.

How Files are Organised

- **Flat**. In this type of file system all files reside at the same 'level' and no concept of directory exists. All files have to be named uniquely. The implementation of a flat file system is simple and it can be useful within embedded systems.

- **Hierarchical**. In this type of file system files are organised over multiple 'levels' through the use of directories. The hierarchical organisation of files and directories often resembles a tree [30], but this is not always the case. For example, any file system implementing the VFS (Virtual File System) can, and should, support symbolic links which can break the acyclic property of trees. Figure 2.4 shows an example of a hierarchical file system. This is the most common type of file system.

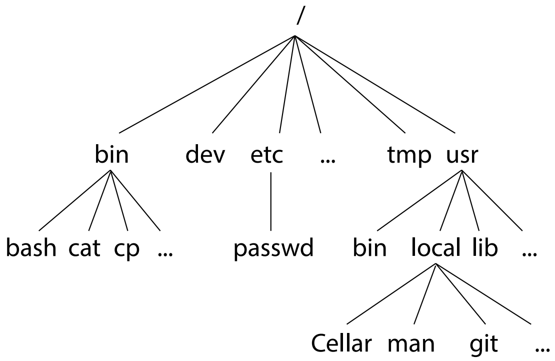


Figure 2.4: Example of a hierarchical file system structure.

[30] A tree is an undirected graph, with no cycles.

How Files and Directories are Implemented

- **Block-based**. Data is stored in blocks which are grouped together by meta-structures, like the *inode* in Unix systems. Block-based file systems are the focus of the rest of this section.

- **Database-based**. A database is used to store files, their attributes, and how they are organised. A database-based file system allows richer queries than traditional file systems. The WinFS [76] and Oracle Database File System (DBFS)[31] are two examples of database-based file systems.

- **Log-Structured**. Data and its attributes are written sequentially to a log, which is implemented as a circular buffer. Log-structured file systems provide good write performance on sequential access storage, have easy-to-implement snapshotting[32], and support easy recovery from failures.

- **Object-based**. The file system uses an object storage device or an object storage service to store its data and metadata. Section 2.2.6 discusses general object storage in more details.

2.2.1.3 The Virtual File System

Unix systems provide a common abstraction for the supported file systems via the *virtual file system* layer (VFS) [77, 78]. The VFS provides two main functions:

- It abstracts different file system implementations by providing a common coherent interface for the OS.

- It allows files to be uniquely identified across the file system implementations (this feature is particularly important for networked file systems).

[31] Introducing the Database File System. Oracle. `https://docs.oracle.com/database/121/ADLOB/adlob_fs.htm` [last accessed on 28/11/2017].

[32] Snapshotting is a technique used to capture a particular state of a system at a given point in time.

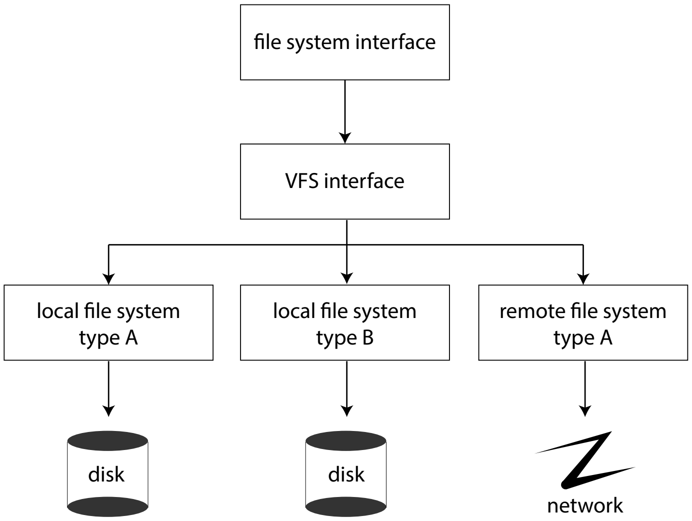


Figure 2.5: Schematic showing the interaction between the virtual file system and the actual file systems. Illustration derived from [31, p. 469]

The diagram shown in Figure 2.5 illustrates how the file-system interface (which enables file operations via system calls such as *open()*, *close()*, *read()*, and *write()*) interacts with a local file system or a remote file system via the VFS. From the perspective of the file-system interface there is no difference between a local or a remote file system, or between two completely different local file system implementations.

The Windows operating systems abstract multiple file system implementations via the *Installable File System* (IFS). This thesis does not cover the implementation details of the IFS, but it should be noted that the IFS provides similar functionalities to the Unix VFS.

2.2.1.4 The File-Control Block

The file-control block (FCB) is the data structure of the file system that contains information about the file [31]. A typical FCB contains the following information:

- File unique identifier.
- File permissions.
- Created time, last accessed time and last modified time.
- Owner, group and access control list (ACL).

- File size.

- Data blocks and/or pointers to data blocks.

In the Unix operating systems the FCB is called inode. The **inode** represents the building block for file systems of the Unix family. Each inode is uniquely identified by its id (*i_ino*) within the file system and there is exactly one inode for each file of the file system. Code 2.3 shows a simplified inode data structure (see Figure 2.6 for a graphical representation of an inode).[33]

```
struct inode {
    umode_t                         i_mode;
    unsigned long                   i_ino;    // identifies the inode
    kuid_t                          i_uid;    // user id
    kgid_t                          i_gid;    // group id
    loff_t                          i_size;
    struct timespec                 i_atime;
    struct timespec                 i_mtime;
    struct timespec                 i_ctime;
    const struct inode_operations   *i_op;
    blkcnt_t                        i_blocks;
}
```

Code 2.3: Simplified inode struct. An example of a full inode struct can be found in the header at the path *include/linux/fs.h* of the Linux kernel.

2.2.1.5 Symbolic and Hard Links

In most modern file systems, the acyclic property of a hierarchical file system is broken by links. Links are "shortcuts" that are used to give the user (or an application) the impression that a file exists in multiple locations. There are two types of links: **hard**

[33] The source code for the Linux kernel is available at the following link: `https://github.com/torvalds/linux` [last accessed on 16/11/2017].

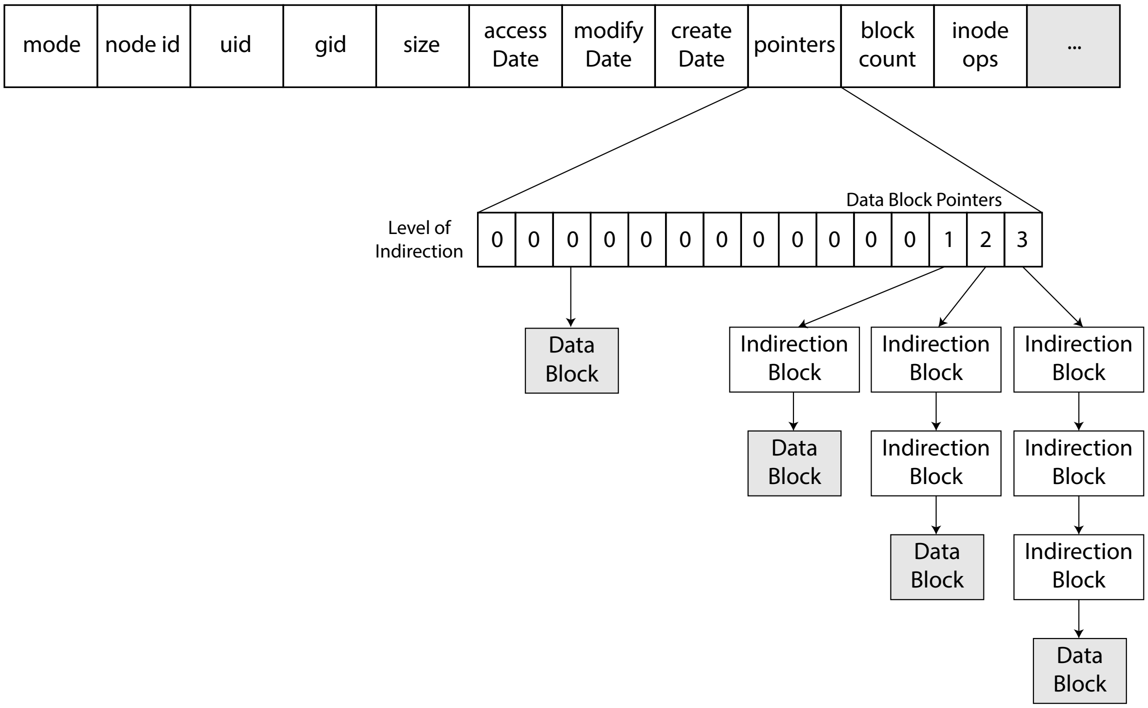


Figure 2.6: Schematic of the inode data structure.

links and **symbolic links** (sometimes referred as symlinks or soft links). The term hard link is used to refer to the linking between a file name in a directory and an inode. In a file system each file has at least one hard link, but multiple and independent hard links are also allowed. Hard links are supported by Unix-like OS as well as the Windows NTFS [79], but not by Windows file systems FAT [80] or ReFS [81]. A symbolic link, instead, is a name pointing to an inode, which itself points to another inode.

The advantages of using hard links is that no difference exists between two hard links pointing to the same inode, so for example if one of the hard links is deleted the other is still valid. If it were a symbolic link, instead, this becomes invalid as it would point to a no-longer existing entity of the file system. Hard links must exist only within the same device and cannot be used for directories, while symbolic links can. In NTFS, links to directories exist, even across different volumes, and are called **junction points** [82].

2.2.1.6 Application Layer File Systems

The file systems that have been seen so far are implemented at the kernel layer, which is a hard task. To avoid writing file systems at the kernel level, system designers can choose to implement file systems in the user space with libraries such as FUSE (Filesystem in Userspace)[34] for Unix systems or Dokany (previously known as Dokan)[35], WinFsp (Windows File System Proxy)[36], and Eldos CBFS (Callback File System)[37] for Windows. Within Unix systems, file systems built in the user space are known as **user space file systems**, while in Windows systems they are sometimes called **user mode file systems**, but for the purpose of this work we introduce a more generic term: **application layer file systems**.

In this section only the FUSE library is going to be taken into account because of its maturity and popularity. Other libraries provide similar properties and functionalities for Unix or Windows operating systems, or both.

FUSE

The FUSE library consists of three main components: a kernel module (fuse.ko), a user space library (libfuse.*) and a mount utility (fusermount). The mount utility allows FUSE-based file systems to be mounted and unmounted.[38] The kernel module mimics a kernel file system and interacts with the VFS. The FUSE kernel module accesses the file system implementation through the user space library, which in turns communicates with the user defined file system. The diagram in Figure 2.7 shows the interactions between the FUSE components and the OS in user space and the kernel space.

[34] libfuse. `https://github.com/libfuse/libfuse` [last accessed on 17/11/2017].

[35] Dokany. `https://github.com/dokan-dev/dokany` [accessed on 24/11/2017].

[36] WinFsp. `http://secfs.net/winfsp/` [last accessed on 24/11/2017].

[37] Eldos Callback File System. `https://sbb.eldos.com/cbfs/` [last accessed on 24/11/2017].

[38] Mounting is the process of attaching a file system to a directory (usually under the root directory */*, or under the directories */Volumes* or */mnt*) and make it available to the system.

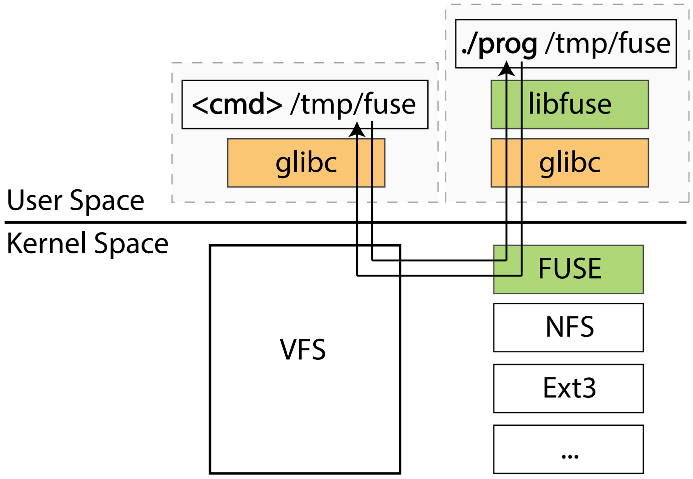


Figure 2.7: Schematics of the FUSE library and its interaction with the VFS through the operating system user space. Illustration derived from [83].

Examples of file systems built with FUSE are:

- GmailFS[39], which uses Gmail to store data.
- MinFS[40], which abstracts the Amazon S3 object store as a file system.
- SSHFS (SSH Filesystem)[41], which provides access to a remote file system via SSH.
- WikipediaFS[42], which allows access to Wikipedia as a file system.
- davfs2[43], which uses FUSE and a network library to provide resources on a WebDAV server as a file system.

2.2.2 Database Systems

“A database is a set of data that has a regular structure and that is organised in such a way that a computer can easily find the desired information” [84]. In other words, a database is a collection of data **records** which are associated by some relevant **properties**, or **fields**, and can be further associated with other records. For example, the database of

[39] GMail Filesystem over FUSE. `https://sr71.net/projects/gmailfs/` [last accessed on 17/11/2017].
[40] MinFS. `https://github.com/minio/minfs` [last checked on 17/11/2017].
[41] SSHFS. `https://github.com/libfuse/sshfs` [last accessed on 17/11/2017].
[42] WikipediaFS. `http://wikipediafs.sourceforge.net/` [last accessed on 17/11/2017].
[43] davfs2. `http://savannah.nongnu.org/projects/davfs2` [last accessed on 24/11/2017].

a registry office will have records about all the people of a specific area, with each person having fields such as: name, surname, data of birth, place of birth, *etc.* The records can then be queried using these fields, for example by querying all the people named 'John' and born before the year 1975.

Over the last decades several database models have been created, with the most relevant being the following.

Flat. This is the simplest type of database model and it consists of a table with fields mapping to records [85]. Flat databases are usually implemented as files. The */etc/passwd*, */etc/group*, and */etc/hosts* files in Unix systems are such examples.

```
##
# Host Database
##
127.0.0.1        localhost
255.255.255.255  broadcasthost
::1              localhost

127.0.0.1        test-server.co.uk
192.168.168.1    external-server.com
```

Code 2.4: Example of an */etc/hosts* file.

Relational. The relational database model is based upon **records** stored in multiple tables and related to each other via special fields known as **keys**. The relationships, constraints and keys of all the tables define the **schema** of the database. Examples of relational databases are: SQLite[44], MySQL[45], MariaDB[46], Microsoft Access[47], and Microsoft

[44] SQLite. `https://sqlite.org/` [last accessed on 21/11/2017].

[45] MySQL. `https://mysql.com/` [last accessed on 21/11/2017].

[46] MariaDB. `https://mariadb.org/` [last accessed on 21/11/2017].

[47] Microsoft Access. `http://office.microsoft.com/access` [last accessed on 21/11/2017].

SQL Server[48].

A record is commonly associated to a row in a table and consists of a list of values. Each value is of a specific type based on the ‘column’ (or field) it belongs. All records of a table have the same number of values and the same value types per column. SQLite, for example, supports five type classes: text, numeric, integer, real, and blob. Table 2.2 shows the associated data types for each of the type classes. More advanced data types are also allowed. For example, MySQL supports data types as: curve, geometry, point, enum, set, *etc.*

Type class	**Data Type**
Text	Character (20) Varchar (255) Varying Character (255) Nchar (55) Native Character (70) Nvarchar (100) Text Clob
Numeric	Numeric Decimal (10,5) Boolean Date Datetime
Integer	Int Integer Tinyint Smallint Mediumint Bigint Unsigned Big Int Int2 Int8
Real	Real Double Double Precision Float
Blob	Blob *no data type specified*

Table 2.2: Data classes and data types for SQLite Version 3.

[48] Microsoft SQL Server. `http://microsoft.com/sqlserver/` [last accessed on 21/11/2017].

Furthermore, some values of the record have special properties and are known as keys. A relational database should support at least **primary** and **foreign** keys. A primary key is the field of the record that marks it as unique within the table. In some cases, a primary key can be composed by multiple fields and it is known as a **composite primary key**. A foreign key is a field of a record that has a corresponding match with a record of another table. Foreign keys are fundamental as they allow relations between records of multiple tables to exist.

Object-oriented. Object-oriented databases are similar to relational databases, but objects, instead of tables, are used to hold the records. Object-oriented databases integrate well with object-oriented programming languages, such as Java, C++, Python, Objective-C or Smalltalk, since they share similar types of abstractions. Hybrids between object-oriented databases and relational databases also exists, such as PostgreSQL[49].

The **object** abstraction consists of attributes, which define the characteristics of the object, and methods, which define the behaviour of the object. An important property of objects is that they consist of data as well as computation.

The object abstraction is used in storage solutions that would not normally be classified as databases, such as orthogonal persistence storage systems [86, 87, 88], which are examples of object-oriented storage systems where objects are persisted for as long as required. Moreover, objects could also be related one another, forming a graph of objects (see below for graph database systems) and providing a better understanding of the data to users.

Document-oriented. A document-oriented model is one where records are stored as **documents** (examples of encodings are: XML, JSON, YAML, or even the PDF document format) and do not have any fixed schema. The capabilities and limitations of a document data value depends on its format. Examples of document-oriented databases are: MongoDB[50], RethinkDB[51], and Couchbase[52].

[49] PostgreSQL. `https://postgresql.org/` [last accessed on 21/11/2017].

[50] MongoDB. `https://mongodb.com/` [last accessed on 21/11/2017].

[51] RethinkDB. `http://rethinkdb.com/` [last accessed on 21/11/2017].

[52] Couchbase. `http://couchbase.com/` [last accessed on 21/11/2017].

```
{
   "_id": 1,
   "name": {
      "first": "John",
      "last": "Bond"
   },
   "age": 43,
   "email": "john@bond.co.uk",
   "profession": "Independent Contractor",
   "partner": 108, // This is a reference value that points to another JSON document
}
```

JSON 2.5: Example of a JSON document data entity.

Graph. A graph database model is designed around the concepts of representing records as nodes of a graph and relationships as edges between nodes. Relationships always have a direction, a type and can also have a weight. Graph databases are very useful when a non-rigid schema is required and semantic queries are involved. Examples of graph databases or supporting a graph database are: Neo4j[53], Microsoft SQL Server (2017 version), and OrientDB[54].

In the **node-edge** abstraction, data is stored within nodes, while edges represent relationships between nodes. The node-edge abstraction is very generic and can be used also between 'documents' that reference each other through reference values. The Resource Description Framework (RDF) [89] data model is also based on the node-edge abstraction.

Key-value. The key-value database model is based on the simple concept of associating a key to a value, the record. A key-value database is also known as dictionary or hashmap. A key-value model is very simple and allows very fast read/write of records at the cost of very expensive joins[55]. Key-value databases are usually in-memory databases with

[53] Neo4j. `https://neo4j.com/` [last accessed on 21/11/2017].

[54] OrientDB. `https://orientdb.com/` [last accessed on 21/11/2017].

[55] A join is the operation of combining records stored in different tables, documents, or objects by common keys (also known as foreign keys in relational databases).

persistence options. Examples of key-value databases are: Redis[56] and memcached[57]. It should also be noted that high-level programming languages, such as Python or Java, have built-in support for key-value in-memory databases within their language (*i.e.*, *dictionary* for Python, *hashmap* and *hashtable* for Java).

Distributed. A distributed database is one where the data is distributed across multiple nodes. Examples are distributed databases are: Cassandra[58], Amazon DynamoDB[59], or Voldemort[60].

2.2.3 Versioning in Storage Systems

Versioning is the ability to track and manage data changes over time. This section explores how data and versions are represented in storage systems and the most common versioning operations.

The concept of versioning is often bound to Versioning Control Systems (VCS), such as SVN [90], Mercurial[61], and Git[62], but it is a widely used technique in computing and used in backup applications, content management systems, and digital asset management solutions too. Section 3.3 describes the most relevant versioning systems in more depth.

2.2.3.1 The Version Metaphors

The *version* metaphor abstracts the data metaphor as it changes over time, so that its evolution over time is preserved. The simplest version abstraction consists of the identifier (or name) and the data of the version. The identifier should be unique across all the versions of the entity. In some systems the identifier is incremental so that it is possible to reconstruct the history of the entity by just looking at the names of the versions (see Figure 2.8a), as in the RCS (Revision Control Software) system [91]. Alternatively, some version abstractions store references to related versions of the entity (see Figure 2.8b), such

[56] Redis. `https://redis.io/` [last accessed on 21/11/2017].

[57] memcached. `https://memcached.org/` [last accessed on 21/11/2017].

[58] Apache Cassandra. `https://cassandra.apache.org/` [last accessed on 22/11/2017].

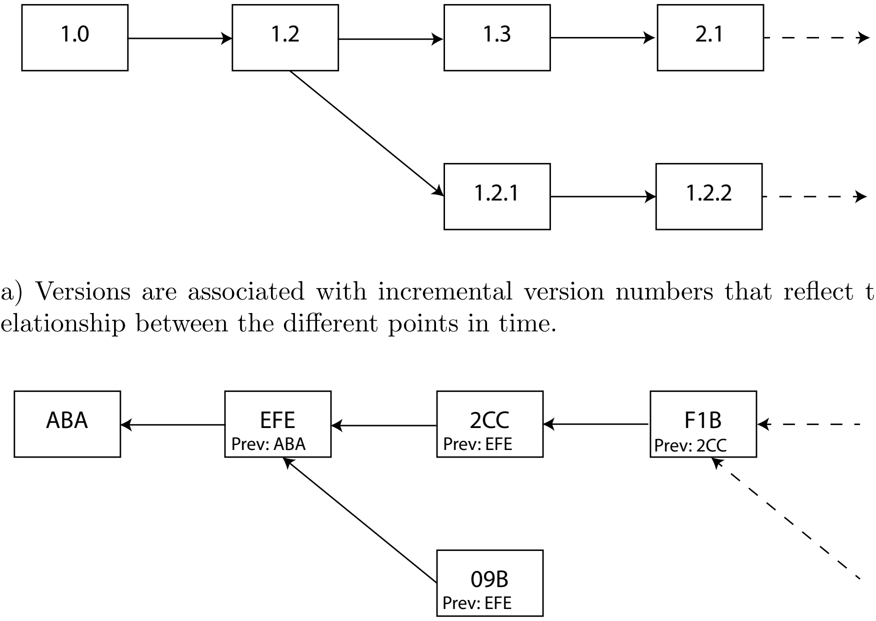
[59] Amazon DynamoDB. `http://aws.amazon.com/dynamodb/` [last accessed on 22/11/2017].

[60] Project Voldemort. `http://project-voldemort.com` [last accessed on 23/11/2017].

[61] Mercurial-scm. `https://mercurial-scm.org/`

[62] git-scm. `https://git-scm.com/`

as in the git [92] and mercurial [93] version control systems.



(a) Versions are associated with incremental version numbers that reflect the relationship between the different points in time.

(b) Versions with unique identifiers and storing references to related (previous) versions.

Figure 2.8: Diagram showing two different ways of organising multiple versions for a given versionable entity. The arrows indicate the relationships between the versions.

2.2.3.2 Versioning Terms and Operations

This section lists the common terminology used for versioning system and throughout this thesis.

- **Repository**. This is where all the versioned content is stored.
- **Local working copy**. This is a local copy of the repository. Its contents are changed until committed to the repository.
- **Remote repository**. This is a repository stored on a remote node, in relation to the local node.

Even thought the operations supported by systems providing versioning can vary, all such systems usually provide the following common operations:

- **Initialisation**. The system initialises a given location within the storage system and starts tracking changes within that location. The initialised location is usually called a **repository** or **depot**. Some systems allow repositories to be located in multiple machines.

- **Snapshot** or **commit**. The current state of the repository is stored by the storage system for later use. Each snapshot is uniquely labelled within the history of the repository.

- **Update**. Copies the contents of a snapshot into the local working copy.

- **Pull**. This is the operation of retrieving the latest snapshots from a remote repository. The first pull of a repository is often associated with the initialisation of the local repository (**clone** operation).

- **Push**. This is the operation of updating a remote repository with the local snapshots.

- **Branch**. The repository is copied locally so that its contents can evolve differently from the original repository. Branches play an important role when multiple end-users collaborate over a repository. When a system does not allow branching, then content is **versioned linearly**.

- **Merge**. This operation involves the merging of two branches. The conflicts originating from a merge should be resolved manually or by the provided tools.

- **Revert**. The system enables end-users to revert the state of the repository to an arbitrary snapshot.

- **Log**. The system provides a list of all the available snapshots (the **history** of the repository).

The operations above, however, may not be supported by all systems that provide versioning. For example, Apple's Time Machine and ZFS do not provide any support for branching and merging.

2.2.4 Networked File Systems

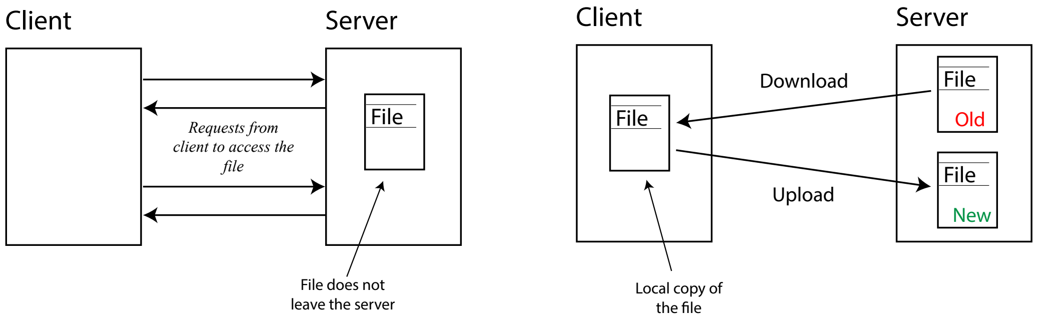
A networked file system is an implementation of a file system that involves a client and one or more servers, with the servers storing and managing the actual file system and the client remotely accessing it through the server and acting upon its files and directories as if it were a local file system. Networked file systems have existed for decades in a variety of forms, from client-server to peer-to-peer file systems. However, networked file systems are not always clear and easy to categorise [94, 95, 96]. In this thesis we propose the following categories:

- **Client-server file systems** (CSFS): one or more clients have access to the file system of a remote server (or multiple servers that appear to be as one).
- **Clustered file system** (CFS): the file system is managed by a pool of multiple servers with a "shared-disk" model. In a "shared-disk" model, the disk of a server is accessible to all other servers. The physical boundaries of a clustered file system are transparent to users.
- **Shared-nothing file system** (SNFS): the file system is built as a pool of multiple servers with a "shared-nothing" model. In a "shared-nothing" model, each server can access only its own disks.

2.2.4.1 Client-Server File Systems

The main goal of a CSFS is to provide remote storage, known as network-attached storage (NAS), while ensuring transparency over the remote data, so that users perceive the remote file system as a local file system. CSFS are characterised, mainly, by the namespace, the service type, and the data access model.

The namespace can be of two types: **per-client**, each client uses its own naming system; and **global**, each client uses the same common naming space. The service type can be either **stateless** or **stateful**. In a stateless service the client's requests to the server are self-contained; while in a stateful service the requests are shorter since the server stores information about the client's requests through time. A stateless service requires more expensive network requests, but its design is simpler to implement, it is resistant to server failures, and the server does not need to keep any in-memory storage. A stateful service, instead, has a more complex design, but it allows cache coherency, performance improvements via caching, and file locking. The data access model characterises the interactions between the components of a CSFS. There are two model types: the **remote access model** and the **upload/download model**. In the remote access model the server is accessed on each file operation (*i.e.*, open, close, read, write, etc.). The remote access model is easy to implement, but its simplicity comes with a network overhead that can result in poor performance for most use cases. The network overhead, however, is compensated by the ability of the model to serve the client only with the data that is needed, which is especially useful for large files. In the upload/download model, instead, the file is cached in the client and upload/download requests with the server are performed asynchronously. Caching the data results in faster IO when operating on the data, compared to the remote access model. Managing big files with the upload/download model is expensive, since the entire file must be moved from/to the server.



(a) Remote access model.

(b) Upload/download model.

Figure 2.9: Diagrams derived from [32, p. 492].

Examples of CSFS are: the Network File System (NFS) [97], the Andrew File System (AFS) [98], and the Microsoft Server Message Block (SMB)/Common Internet File System (CIFS) (CIFS is a type of SMB protocol) [99, 100]. The NFS provides a stateless service, implements a remote access model, and has a per-client namespace.[63] The AFS implements a stateful service, supports the upload/download model, has a global namespace, has built-in data replication mechanisms, and a built-in authentication protocol. SMB, which has seen improvements over the years, implements a stateful service and supports the remote access model with the ability to cache data locally, and a namespace that is local to the nodes of the system.

2.2.4.2 Clustered File Systems

A clustered file system is defined as a non-local file system built as a pool of multiple servers with a "shared-disk" model. A CFS uses a storage-area network (SAN) to allow multiple servers to share the same storage at the block level. The diagram in Figure 2.10 is a simplified example of two servers that communicate with each other via a private connection and share the same disk. The two servers, however, appear as one to the client which accesses it through a public cluster connection.[64] A CFS must adopt a concurrency control model to avoid multiple clients modifying the data inconsistently.

A clustered file system can present some important design issues:

- Single point of failure (SPOF) on the shared disk.
- Concurrency control model's complexity is directly proportional to the number of clients accessing the CFS.

Examples of clustered file systems are Lustre [106], the Oracle Cluster File System (OCFS2) [107], and GFS2 [108].

2.2.4.3 Shared-Nothing File Systems

A distributed system is commonly defined as "a collection of independent computers that appears to its users as a single coherent system" [32]. In a distributed system we often have multiple servers under a shared-nothing (SN) model, where nodes are independent

[63] New versions of the NFS have been released over the years, each improving the design of the predecessor. These are the NFSv3 (RFC 1813 [101]) and the NFSv4 (RFC 3010 [102], RFC 3530 [103], RFC 7530 [104]).

[64] A public cluster connection can also be setup through one of the cluster nodes, which is then called the master node.

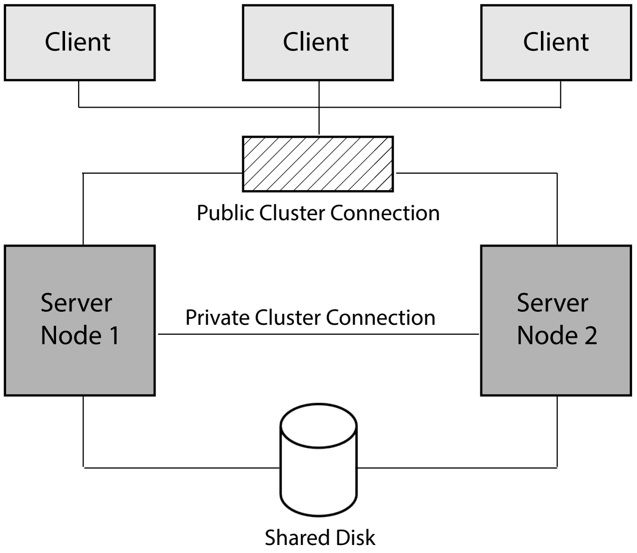


Figure 2.10: Clustered File System schematics, derived from [105].

in order to eliminate any SPOF, favouring fault-tolerance and self-healing. A SNFS is one application that can be run over a distributed system and can be defined as *a file system shared across distributed independent components of the system.* Shared-nothing file systems are usually known as "distributed file systems", but the SNFS naming better reflects the architecture of the file system, while the DFS naming can be confused with the CSFS and CFS.

SNFS can be either **asymmetric** or **symmetric**. In a symmetric SNFS, all nodes are equal (*i.e.*, they have the same potential capabilities) and they are known as peers. The mechanism behind this design is simple: if one peer needs data, other remote peers are questioned about it until the data is found and retrieved. Symmetric SNFS are also known as peer-to-peer distributed file systems (P2P-DFS). In an asymmetric SNFS, instead, some nodes are more important than others or simply have different roles, irrespective of their importance. This architecture is also known as the client-server architecture, where there are nodes serving the data (*i.e.*, servers) and nodes requesting it (*i.e.*, clients).

Understanding the client-server architecture is fundamental to understand not only asymmetric SNFS, but also symmetric SNFS. In fact, even though some nodes may play both the client and the server roles (this is especially true in the case of P2P systems), the overall system can still be observed and studied as a client-server architecture.

In any client-server SNFS there are two primary services: the **name server** and the **cache manager**. The name server resolves the name-location mapping, while the cache manager stores data locally, so to avoid future network requests and reduce drastically data access, resolving inconsistencies between nodes (Figure 2.11 shows how the cache manager is used to reduce data access latency).

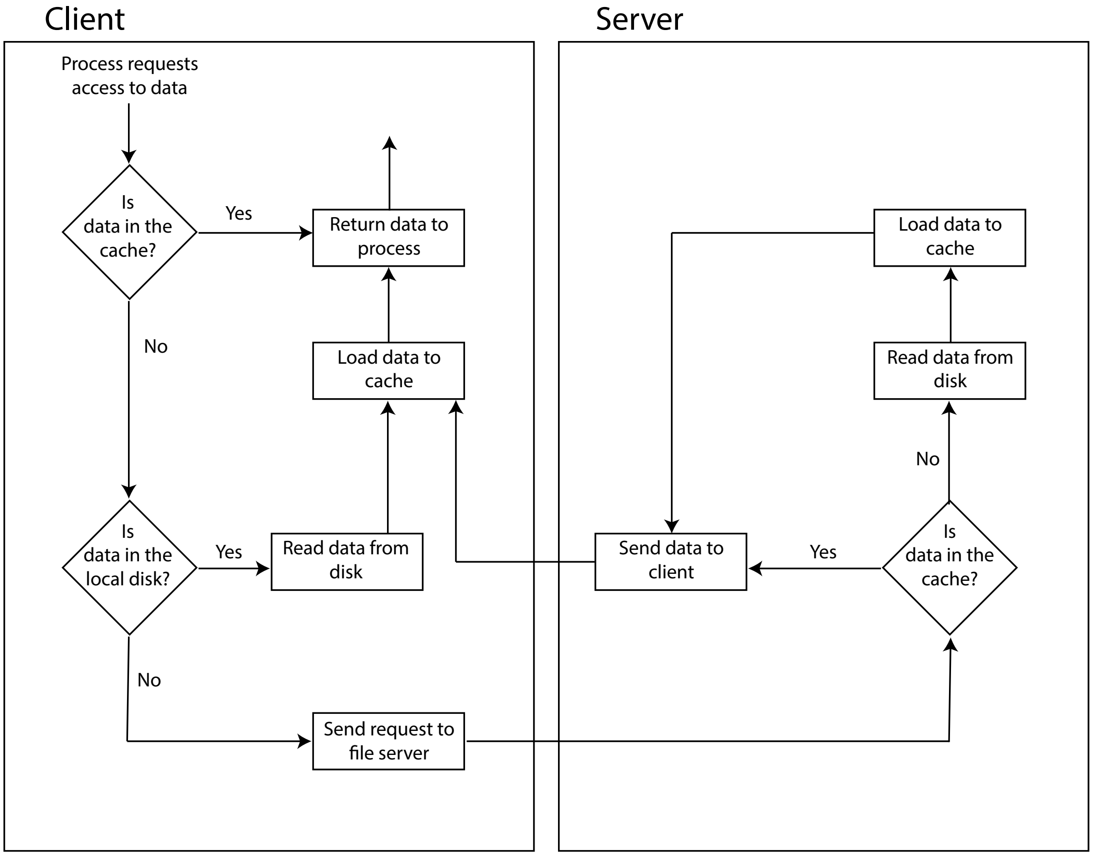


Figure 2.11: This diagram shows the role of the cache manager in the client and the server within a shared-nothing file system. Diagram derived from [109, p. 201]

The size of a SNFS is usually described in terms of the number of nodes composing it, which can variate from a minimum of two nodes to millions and more. Very large SNFS are usually called *global file systems*. Examples of SNFS are the Hadoop File System [110], the Google File System [111], Oceanstore [112], and the InterPlanetary File System (IPFS) [113].

2.2.5 Cloud Storage

Cloud computing is a recently coined term used for distributed systems that provide on-demand service. The proper definition of *cloud* has been, and still is, debatable [114, 115, 116, 117, 118]. In the seminal book "Cloud Computing - Principles and Paradigms", by Buyya *et al.*, the authors suggest that irrespective of what definition of *cloud* one decides to use, cloud computing should have the following properties:

1. **Pay-per-use**: users should be able to easily release resources and be billed only for the services used.

2. **Storage and computation elasticity**: users should have the illusion that the cloud service provides infinite storage and computation capabilities at any time. The provision of resources should adapt automatically to the load on the service or application.

3. **Self-service interface**: the services provided should be easy to use, set-up and manage, without any human intervention from the cloud provider.

4. **Abstraction or virtualisation of the resources**: users should not be aware of the hardware and network infrastructure and complexity of the provided service [22, p. 4; pp. 16-17].

Cloud services are divided into three categories based on the abstraction level of the service provided: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [22].

IaaS provides high-level APIs over the low-level components of the cloud infrastructure. Examples of IaaS storage are Amazon S3 [20], Amazon DynamoDB [119], DigitalOcean Spaces [120], and Rackspace Cloud Object Storage [21].

PaaS, instead, consists in providing users with a platform to develop, run, and/or manage their applications. In PaaS, storage is provided as part of the platform and examples are AWS Elastic Beanstalk [121], Heroku [122], and the Google App Engine [123] which allow easy deployment of applications.

Finally, SaaS are applications that ‘live’ in the server side. Well-known examples of SaaS storage applications are Dropbox [18], OneDrive [19], Google Drive [124], and Backblaze [125].

2.2.6 Object Storage

In object storage solutions, data is represented as objects, which are composed of three components: the data itself, a unique identifier and any associated metadata [126, 127]. The unique identifiers used for objects are part of a flat global namespace, making object storage very scalable. The flat structure also makes data creation and retrieval easy and efficient operations, especially for very large amount of unstructured data. On the other hand, however, applications relying on object storage need to maintain a naming service to resolve human-readable names to the unique identifiers of objects.

Object storage is particularly common in IaaS cloud storage services, such as Amazon S3 or Rackspace Cloud Object Storage.

Literature Review

This chapter provides a literature review of the related work relevant to the design of a storage system that reflects the properties presented in the *Introduction* Chapter. The literature review is organised by types of storage systems, as defined in the previous chapter. It should be noted that some of the systems presented in this chapter can belong to more than one type of storage system.

3.1 File Systems

File systems are the most commonly known and used storage abstractions of the last six decades. The concepts introduced in Section 2.2.1 are the foundation of most file systems. This section briefly summarises the most relevant concepts of the state of the art in file systems, while a more thorough literature review for non-local storage systems, from networked file systems to P2P storage, is reported in the rest of the chapter.

The *file* metaphor, as defined in Section 2.2.1.1, groups together data, attributes, and operations. The attributes are metadata that provide access to the data, access control, and additional information to the operating system. The basic operations over files are: file creation, data writing, data reading, deletion, and file truncation. Section 2.2.1.1 also defined the *directory* metaphor, an entity that aggregates files and/or other directories.

Harper *et al.* [14] question what a file is, stating that the file abstraction is not properly understood because it has been overused across many different systems and use cases over the years. The authors propose to challenge the file abstraction and rethink the attributes and actions that are related to it. In particular, their work questions the meaning of actions such as save, copy, and delete within systems that are distributed across multiple end-points and users. The role of metadata, whether intrinsic or extrinsic to the file, is also not very well understood, while being very important to users. For example, should a copy operation be performed only on the data or also on its metadata? Who should be allowed to perform copy operations? And how can multiple copies be perceived as being the exact same identity, but replicated across multiple locations?

This section describes two different approaches that have been attempted over the years to improve the file metaphor: extending the support for intrinsic metadata (see Section 2.1.3 for the different types of metadata) and tagging files.

3.1.1 Extended Attributes Support

Extended attributes are structured metadata associated with the file structure and used to provide richer information about the file. Mogul [128] was one of the first to propose a mechanism to enrich files with additional properties, which were stored in an *ad-hoc* database that could answer complex queries about the files, such as "what files were modified last month?" and "what programs depend on the graphics package library?". Extended attributes are now supported by a large number of file systems (*e.g.*, ext2-3-4, ReiserFS, Btrfs, Lustre, FAT, NTFS, *etc.*) to record additional information about files, such as their provenance, and to improve file searches. File search applications, such as Spotlight[65] and Alfred[66], use extended attributes to provide better search features.

An alternative approach to store extended attributes within the file structure is adopted by Apple's HFS and HFS+, where files consist of two components: the data fork and the resource fork [129, pp. 121-125]. The former is the actual file's data, while the latter contains any additional structured metadata that can be used by users and/or applications to understand better the data or to improve some of the system features.

A third alternative to model files with richer metadata consists of creating metadata indices. The Be File System [130] and the Haiku File System [131], for instance, implement

[65] Apple - Use Spotlight on your Mac. `https://support.apple.com/en-gb/HT204014` [last accessed on 05/04/2018].

[66] Alfred - Productivity App for Mac OS X. `https://alfredapp.com/` [last accessed on 05/04/2018].

an indexing mechanism at the kernel level to record any metadata associated with files. Indices can later be used to perform queries on files. Both systems also support 'live' queries, where results are updated automatically over time without the users having to trigger the query explicitly. Two disadvantages of this approach are that the process of indexing can impact significantly the overall IO performance of system and that the number of indices may grow very quickly as more metadata attributes are recorded.

3.1.2 Tagged Files

A second approach to enhance the file metaphor is via tagging [132]. Tagging is also supported by MacOS [133], but as a component of the OS external to the file system implementation. A tag is a named key that can be associated with a file or a directory and that aids users to easily find and aggregate files. Moreover, multiple tags can be associated with a given file or directory.

The Resource Description Framework (RDF) is a metadata model that allows resources to be related — or tagged — by specifying triples of the form *(subject, predicate, object)* [134]. In RDF, resources — subjects and/or objects — are related by the predicate. A typical RDF example is "The sky has the colour blue", where "The sky" is the subject, "has the colour" the predicate, and "blue" the object. When applied to a file system, files and directories can be subjects, while predicates and objects can be literals or references to other files and directories (for objects only). The pStore file system [135], for example, uses RDF to record schema-less metadata about files and directories.

3.2 Networked File Systems

Section 2.2.4 defines networked file systems and the different categories they can belong to: client-server file systems, clustered file systems, and shared-nothing file systems. Client-server and clustered file systems are designed to work within distributed systems of small/medium size. For the purpose of this thesis, only shared-nothing file systems (SNFS) will be described in detail. SNFS scale over thousands of nodes and are capable of managing petabytes, or even exabytes, of data. The systems taken into account are the Hadoop File System [110, 136], the Google File System [111], and the metadata management approach used in GlusterFS [137].

Networked file systems can also be used to provide cloud storage, as discussed later in Section 3.4.

3.2.1 The Hadoop File System and Google File System

The Hadoop File System (HDFS) and Google File System (GFS) are shared-nothing file systems (see Section 2.2.4.3) designed to serve very large unstructured data with low latency, high availability, high scalability, and high resiliency. The GFS is a proprietary solution originally designed by Google in 2003, which then inspired the development of the open source HDFS (Apache License 2.0). The HDFS and GFS are designed to be used by other applications and/or services, rather than by user machines.

3.2.1.1 Files, Metadata, and Blocks

Both HDFS and GFS implement the file and directory metaphors used by standard file systems (see *Background* Chapter) in similar ways. Files are typically in the order of gigabytes or terabytes and they are chunked into smaller **blocks**[67] (by default the maximum size of a block is 64MB on GFS and 128MB on HDFS), which are uniquely identified by IDs and replicated across multiple storage nodes. In the HDFS, blocks are stored together with their corresponding checksums, so that the integrity of the blocks can be verified. On the GFS, a block is chunked into smaller sub-blocks of 64KB and checksums are stored for each sub-block.

The file name, the number of replicas stored, and the IDs of the blocks constitute the **metadata** of a file. Unlike common Linux file systems, directories are not inodes mapping content names to other inodes, but they are simply represented by the files' names.

3.2.1.2 The Client, the Coordinator, and the Datastore

The architecture of the HDFS and GFS systems is based on three types of nodes:

- The **Client** is any node that wants to write or read data to/from the file system.

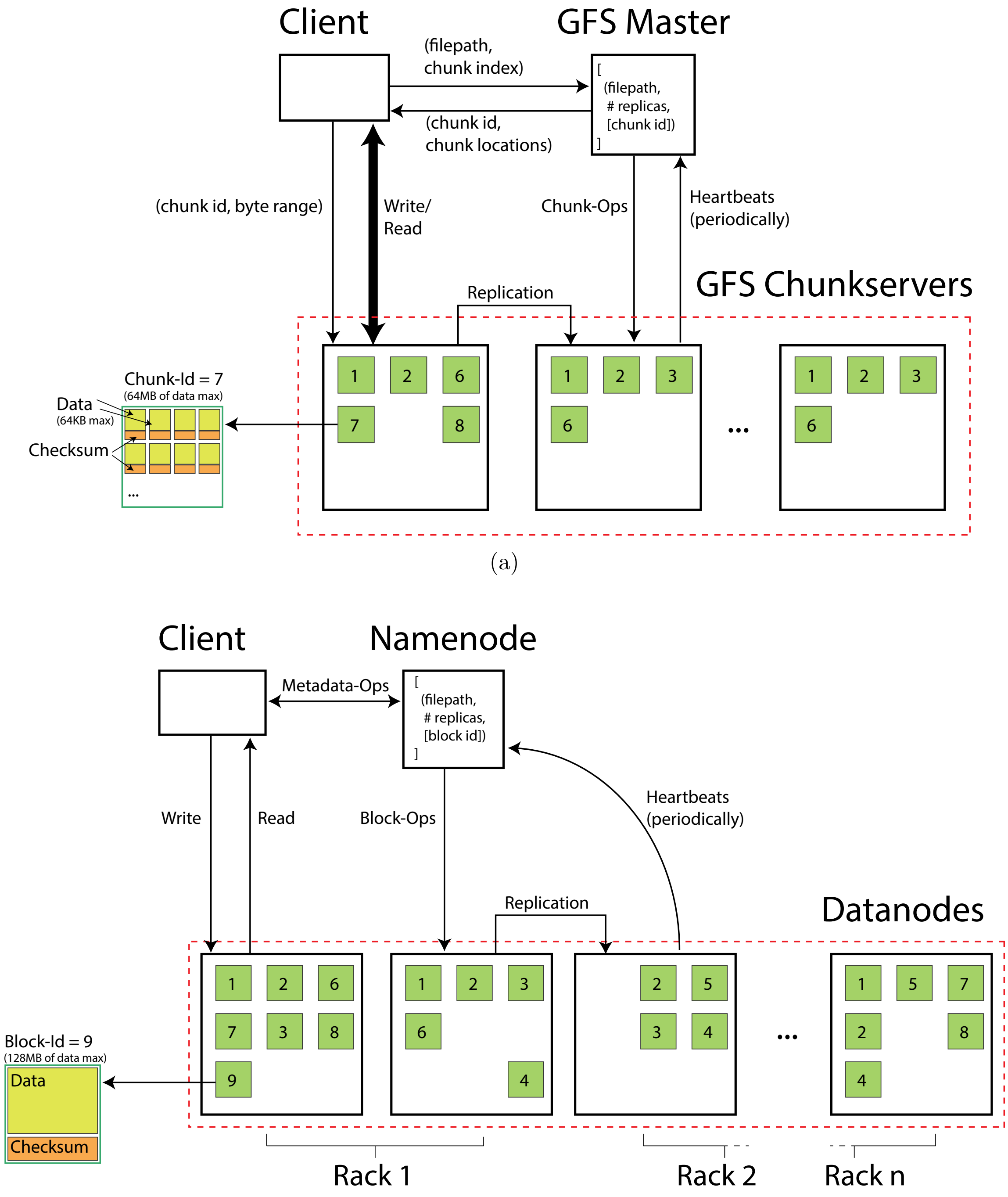
[67] Blocks are called *chunks* in the GFS.

- The **Coordinator** — *Master* in GFS and *Namenode* in HDFS — is responsible for storing metadata information about files, such as their filename, the blocks they are made of, the number of replicas, and where replicas are stored. The coordinator is also responsible for handling access control over the data and for monitoring the status of the datastore nodes. The coordinator triggers any repairing mechanisms when datastore nodes fail.
- The **Datastore** — *Chunkserver* in GFS and *Datanode* in HDFS — is responsible for storing the data blocks.

Figures 3.1a and 3.1b illustrate the architecture components and their types of interactions for the GFS and the HDFS, respectively. There are some important design choices that allow HDFS and GFS to scale to hundreds of thousands of machines and to store petabytes of data:

- The metadata managed by the coordinator is kept small, so that it can be stored in memory and provide low latency access to client and datastore nodes. The HDFS and GFS are meant to be deployed on networks with very low latency, such that even the node disk latency can have an impact on the overall performance.
- The client writes data directly to the datastore nodes, so that the coordinator does not become a bottleneck.
- Replication is handled by the datastore nodes, reducing the communication with the coordinator, which could become a bottleneck in the system.
- The coordinator acts as a load balancer for the datastore nodes.
- Blocks, or chunks, are stored together with checksums, so that the integrity of the data is verified before it is returned to the clients.

The GFS was upgraded around the year 2010 under the name of Colossus [139, 140], but only limited information about its design is known, since no related papers have been



(a)

(b)

Figure 3.1: (a) GFS architecture - diagram derived from [111]; (b) HDFS architecture - diagram derived from [138].

released yet. In GFS, the coordinator is a single point of failure (SPOF), while Colossus supports multiple coordinator nodes, thus having no SPOF. Moreover, Colossus supports erasure coding by implementing Reed-Solomon (see Section 2.1.6.2), so that storage space is reduced while having better resiliency.

New versions of HDFS have been released over the last decade [141]. The new versions of HDFS have introduced two main features to store replicas more efficiently and provide better resiliency: erasure coding over the stored blocks and multiple coordinator nodes, to avoid the coordinator being a single point of failure. By default, HDFS implements the Reed-Solomon erasure coding, but more efficient coding techniques can also be used [54].

One of the main functions of a shared-nothing file system, like HDFS and GFS, is to store metadata and data separately, so that the system does not have bottlenecks on the metadata end-point and can operate with very large quantities of data. This also results in simpler metadata management. However, the amount of metadata increases with the amount of data stored in the system, such that eventually multiple coordinator nodes may be needed. Handling multiple coordinators increases the complexity of the system and the time for clients to retrieve metadata relative to a given file.

3.2.2 GlusterFS

GlusterFS is an open source (GPLV2 and LGPLV3+) [137] shared-nothing file system designed to manage small-medium size files, and usually used to provide cloud storage services. One of the main architectural features of GlusterFS is its *elastic hashing algorithm*, which allows content to be distributed without the need of a metadata service. Thus, GlusterFS does not maintain any index to keep track of where files are stored. The elastic hashing algorithm consists of deterministically determining the location of a file based on the hash of its path, as shown in Figure 3.2. The algorithm is said to be elastic because it can adapt to changes in the number of nodes in the network. Replication and content reliability are achieved via RAID and the abstraction of the physical storage via logical storage volumes.

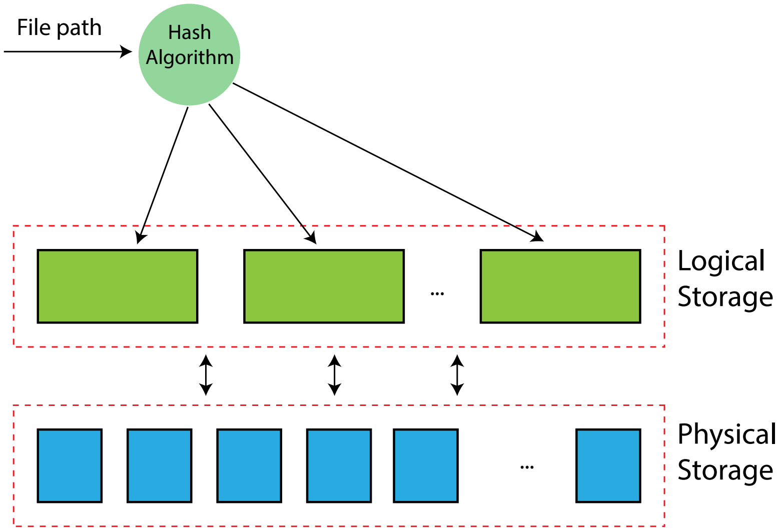


Figure 3.2: GlusterFS architecture. The diagram shows how files are distributed based on the hash of their path.

3.3 Versioning in Storage Systems

Versioning is the process of tracking and managing data as it changes over time. This section explores how different storage systems or versioning tools provide versioning. The concepts and terminology introduced in Section 2.2.3 are necessary to understand the mechanisms and systems presented below.

3.3.1 Manual Data Versioning

The most basic versioning technique consists of manually creating file duplicates, each representing a different version (see Figure 3.3). Manual data versioning is easy to achieve, understand, does not require any additional tools, and is also used by most end-users. Manual versioning also presents some important limitations:

- The relationship between versions is not always clear and it can become hard to understand how content has evolved over time.
- The naming convention is arbitrary, error-prone and hard to adapt to future requirements.
- It is hard to share versions with other users while adopting the same versioning

semantics.

- There is no clear rule on the granularity of the versions.

Version control systems, such as git or mercurial, are better solutions since they have been designed specifically to version content (see Section 3.3.3).

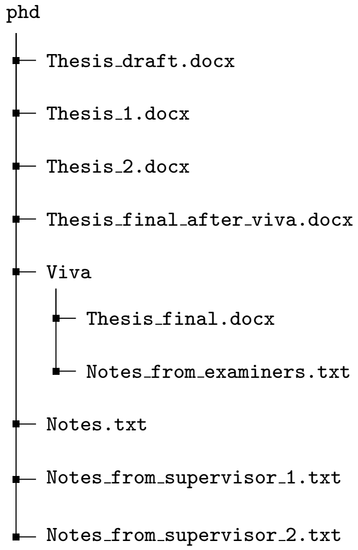


Figure 3.3: Example of manual data versioning.

3.3.2 Versioning in Backup Applications

Backup applications, or services, are solutions that store and manage data so that users can recover previous versions of their data and/or recover lost content if the local storage fails. This section discusses data versioning within backup applications through Apple's Time Machine, which is discussed in more detail below.

Some cloud storage backup solutions are Tarsnap[68], Backblaze[69], and iDrive[70]. Data backups can also be achieved using data synchronisation applications, such as rsync[71] and Syncthing[72].

[68] Tarsnap - Online backups for the truly paranoid. `https://tarsnap.com` [last accessed on 22/04/2018].

[69] Cloud Backup: Easy, Secure Online Backup - Backblaze. `https://backblaze.com/cloud-backup.html` [last accessed on 22/04/2018].

[70] IDrive Cloud Backup. `https://idrive.com/` [last accessed on 22/04/2018].

[71] rsync. `https://rsync.samba.org/` [last accessed on 20/03/2018].

[72] Syncthing. `https://syncthing.net` [last accessed on 16/03/2018].

3.3.2.1 Apple's Time Machine

Time Machine is a backup solution built-in with Apple's operating system since Mac OS X Leopard (version 10.5) [142]. Time Machine backs up the local file system periodically to external disk drives.[73] Time Machine provides versioning by storing incremental backups of the local disk. At each snapshot, files and folders are copied to a backup storage unit, while maintaining the same hierarchical structure of the local file system. Snapshots are stored in folders that are named as following:

```
yyyy-MM-dd-HHmmss
```

Files that do not change across the backups are de-duplicated by creating hard-links to the original files, as shown in Figure 3.4.

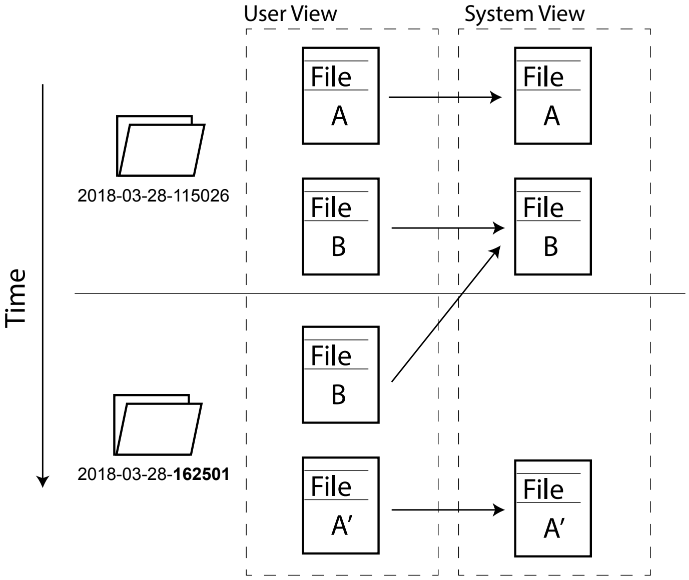


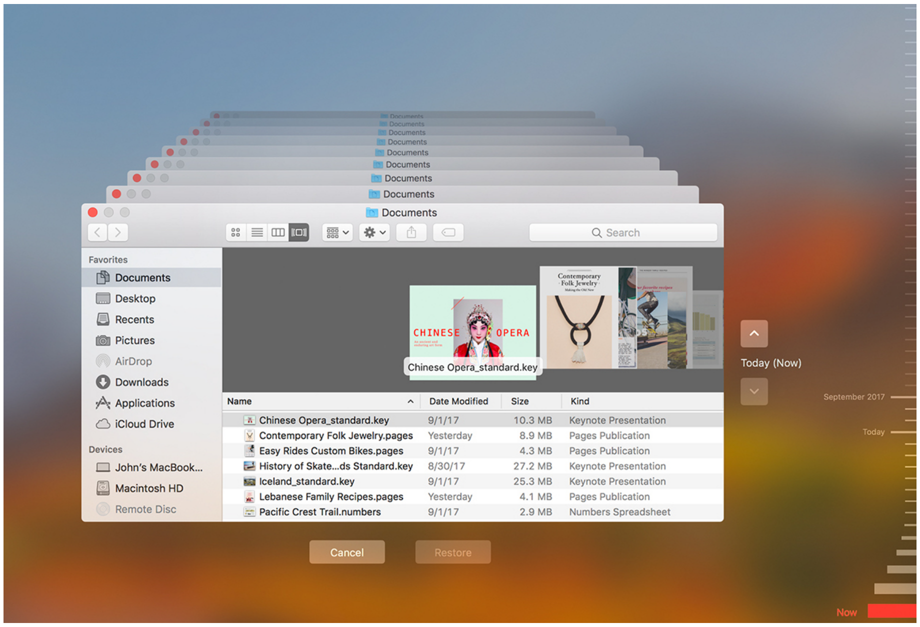
Figure 3.4: Time Machine's storage model. The file B in the second backup (user-view) is a hard-link to the file B as represented in the system view.

When no external disk drive is connected to the machine, Time Machine creates temporary backups that are stored on the local disk drive. These temporary backups are eventually copied to the external disk drives. Time Machine also deletes intermediate backups over time, so that space can be saved over the backup disks.[74]

[73] External storage units can also be connected over the network.

[74] Time Machine keeps all hourly backups of the last 24 hours, daily backups of the last month, and weekly backups until the disk is full. Then, Time Machine keeps deleting the oldest backups.

In addition, Time Machine provides a user friendly interface to interact with the backups and revert the state of a particular file, folder, or the entire local disk to an arbitrary snapshot (see Figure 3.5).



Name	Date Modified	Size	Kind
Chinese Opera_standard.key	9/1/17	10.3 MB	Keynote Presentation
Contemporary Folk Jewelry.pages	Yesterday	8.9 MB	Pages Publication
Easy Rides Custom Bikes.pages	9/1/17	4.3 MB	Pages Publication
History of Skate...ds Standard.key	8/30/17	27.2 MB	Keynote Presentation
Iceland_standard.key	9/1/17	25.3 MB	Keynote Presentation
Lebanese Family Recipes.pages	Yesterday	4.1 MB	Pages Publication
Pacific Crest Trail.numbers	9/1/17	2.9 MB	Numbers Spreadsheet

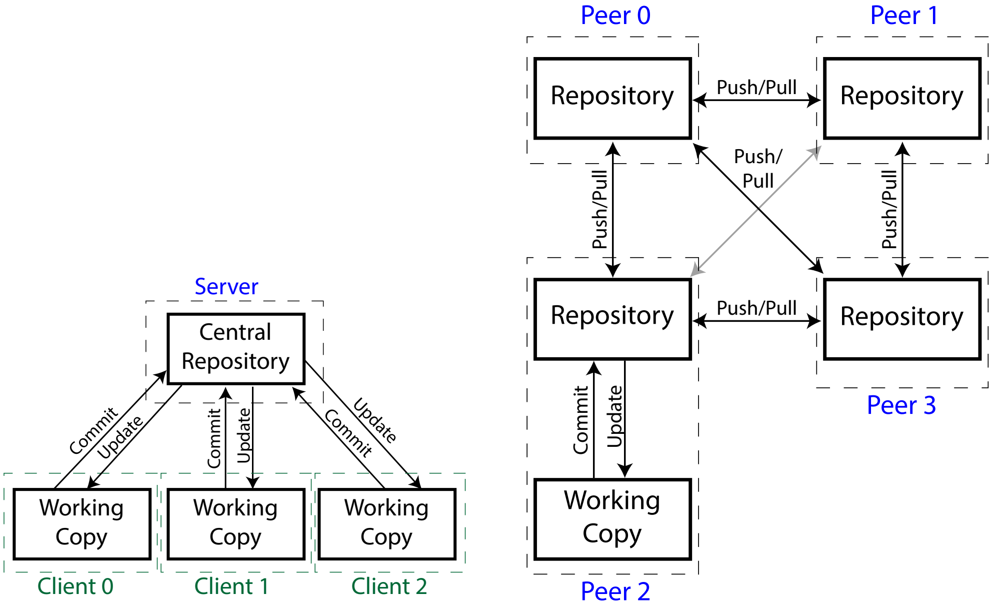
Figure 3.5: Screenshot of the Time Machine's user interface (see *Image Credits*).

3.3.3 Version Control Systems

Version control systems (VCS) are data management systems designed to track versions of a particular repository — often consisting of source code — over time. This subsection describes the difference between centralised and distributed VCS, introduces the Merkle data structure, which has become an important element of many VCS, and then focuses on the two most popular distributed VCS: git and mercurial.

3.3.3.1 Centralised and Distributed Version Control Systems

One way of categorising VCS is based on their architecture, which can be either centralised or distributed. The architecture of **centralised** VCS consists of a single central repository, with clients managing a partial copy of the repository and committing/updating changes from/to the central repository only (see Figure 3.6a). Examples of centralised version control systems are the CVS [143] and SVN [90] systems. In a **distributed** VCS, there is not a central repository acting as the only source of truth, but rather all the actors, or peers, involved have a full copy of the repository and can pull/push their changes from/to any other peer (see Figure 3.6b). Often, however, peers agree upon having one of the repositories acting as the main source of truth (*e.g.*, GitHub[75], BitBucket[76], or GitLab[77] are all services providing support for git and/or mercurial and often act as the central repositories for projects). Examples of distributed version control systems are git, mercurial, and bazaar [144].



(a) Centralised VCS.

(b) Decentralised VCS.

Figure 3.6: Diagram showing the architecture of centralised and decentralised version control systems. All peers in (b) can have a working copy, as shown for Peer 2.

In distributed VCS, end-users can commit changes locally, while in centralised VCS the changes have to be synchronised with the central repository. As a result, in a distributed VCS it is possible to operate over a repository independently of all the other peers involved, except for the push/pull operations that require peers to communicate. However, managing very large repositories - in terms of bytes stored and/or number of files managed - is more efficient in centralised VCS, since the clients manage only the relevant content of the

[75] GitHub. `https://github.com` [last accessed on 23/04/2018].
[76] BitBucket. `https://bitbucket.org/` [last accessed on 23/04/2018].
[77] GitLab. `https://about.gitlab.com/` [last accessed on 23/04/2018].

repository, while the same is not true in distributed VCS.

For the purpose of this work, only decentralised VCS will be discussed further.

3.3.3.2 The Merkle Tree

A Merkle tree is a tree data structure in which leaf nodes contain the hash of a given data block and internal nodes contain the hash generated by hashing the hashes of their children, as shown in Figure 3.7 [145, 146]. A Merkle tree has three important characteristics:

1. Its nodes are content-addressable via their hashes. As a result, nodes can be distributed and accessed independently of their locations.

2. The hash of a node can be used to verify the integrity of the content it refers to, which can be either data blocks or other nodes.

3. Nodes can be de-duplicated if they are addressable by the same hash values.

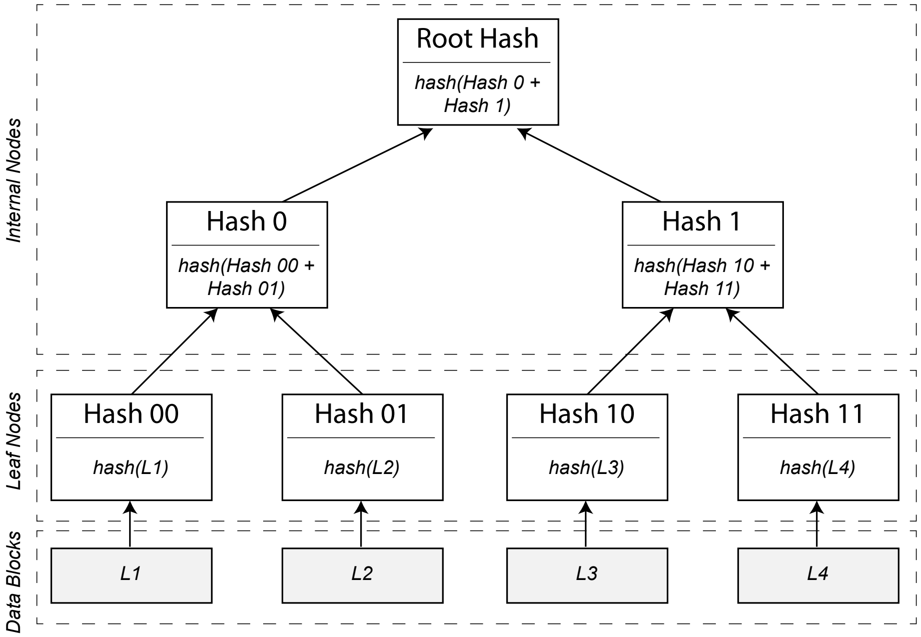


Figure 3.7: Example diagram of a Merkle tree. The arrows indicate the input data used to calculate the hash of a node.

3.3.3.3 Git

Git is an open source (GNU GPL v2) distributed VCS designed primarily to manage source code, but it can be used to control versioning for any type of file [92].

The Git Data Model

The git model consists of two main data structures: a **mutable index**, storing all the relevant information about the working directory, and a collection of first-class **immutable objects**, which evolves over time as content from the working directory is committed. The immutable objects can be of four types and constitute the core of the git data model (see Figure 3.8):

- The **blob** (binary large object) object stores the data of a particular version of a file. The blob consists of the bytes that make up the file plus some metadata necessary to identify and read the blob.
- The **tree** object aggregates blobs and other trees together by reference. A given snapshot of a repository is represented by a single tree.
- The **commit** object links trees together forming the history of a repository. A commit stores a references to a given tree (snapshot), a reference to zero or more previous commits, a timestamp, a string for the author and the committer (*e.g.*, this is usually an email address)[78], and an arbitrary message.
- The **annotated tag** is an object containing a reference to a commit of the repository. Annotated tags are used to label and record particular snapshots of the repository.

All objects described above are identified by a SHA-1 hash[79], which is derived from the contents of the object. The objects are linked together by reference and form a Merkle

[78] Chacon and Straub define the author and committer as following: “the author is the person who originally wrote the work, whereas the committer is the person who last applied the work” [147].

[79] The git community is currently working on transitioning git to use a stronger hash function as suggested here: `https://github.com/git/git/blob/0afbf6caa5b16dcfa3074982e5b48e27d452dbbb/Documentation/technical/hash-function-transition.txt` [last accessed on 20/03/2018]

tree. The objects of the git model are content-addressable and their hashes can be used to verify the integrity of the repository.

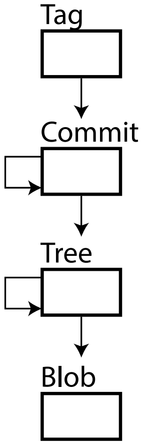


Figure 3.8: Git data model.

Second-Class Objects

Second-class objects are entities derived from the model described above. In git, second-class objects are used to annotate the git model, thus enabling better navigation of the Merkle tree. For instance, the *branch* object maps the name of branches to the relevant commits in the model. The *head* is a reference to the branch a user is currently on. Annotated tags also have their respective second-class objects, so that they can be easily found and retrieved.

Git Internals

Figure 3.9 shows how the content versioned by git is stored internally. First-class objects are stored all in the same location, under .git/objects/, while second-class objects are stored under .git/refs as references to commits or annotated tags.

Git uses three techniques to efficiently store objects:

- **Compression**. All objects are compressed to save space.
- **Chunking**. Large files are stored into multiple chunks (*i.e.*, blobs) and grouped together using trees. As a result, when a large file changes only the changed chunks are stored, while all other chunks are de-duplicated.

- **Packing**. Most objects, including blobs, are smaller than the file system block size (blocks are usually of 4KB, 8KB, or 16KB). Thus to avoid wasting block space, objects are packed together into a single file.

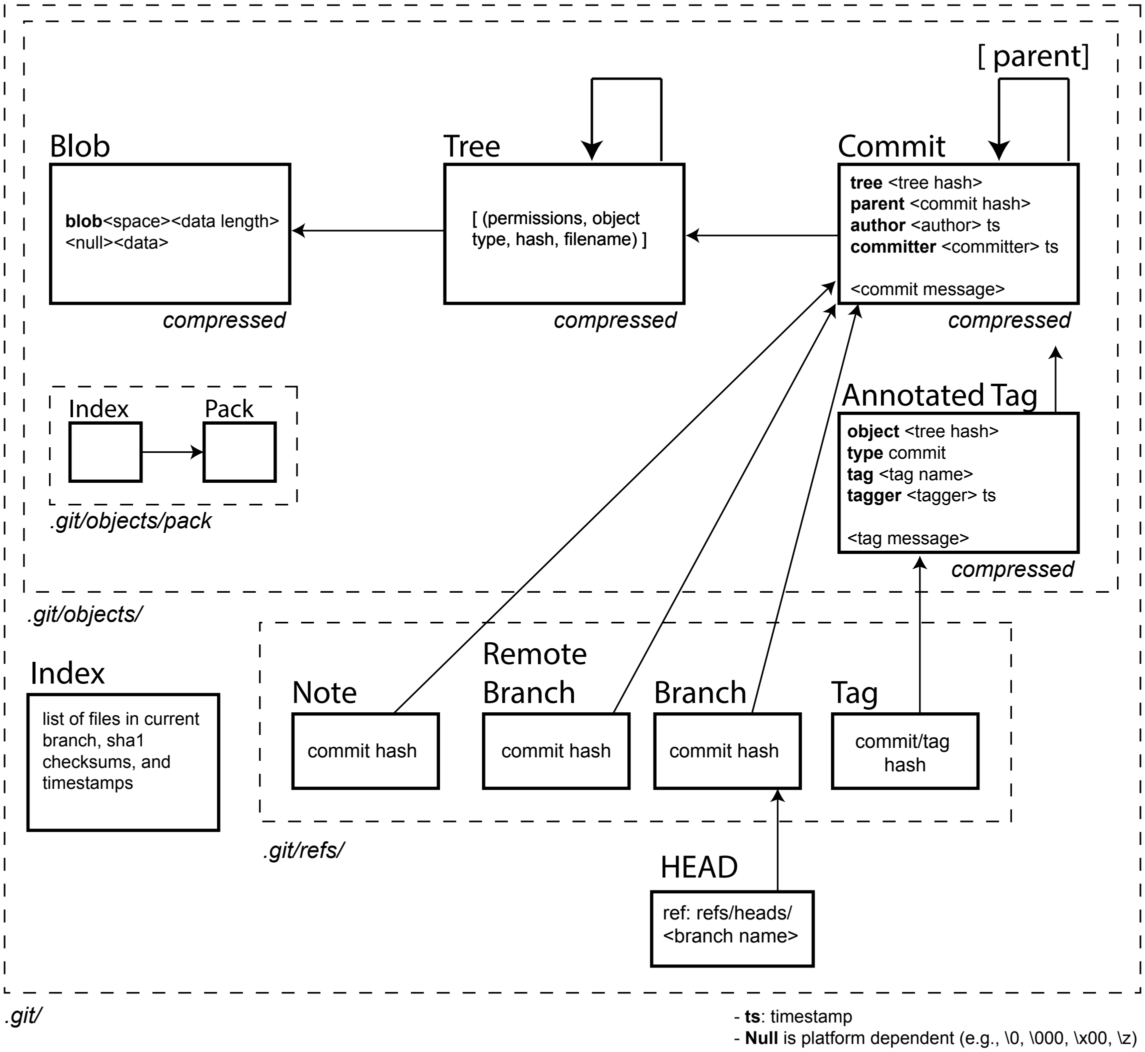


Figure 3.9: Git internal model. Each git repository has a .git/ directory. This diagram shows how and where the first-class and second-class objects of the git model are stored within the ./git directory.

The Git Notes Model

Git provides support for metadata via the commit objects, described earlier, and the git notes model. The git notes model allows users to write metadata as notes and associate them with arbitrary commits of the repository, without changing the history of the repository. A note is an arbitrary sequence of bytes — usually text — that end-users use to describe a commit. The notes model uses the first-class objects of the git data model described earlier, but in a slightly different way (see Figure 3.10):

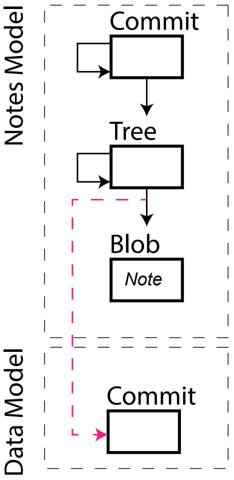


Figure 3.10: Git notes model.

- The **blob** object contains the actual note.
- The **tree** object aggregates all the notes at a particular time in the repository history. Each entry in the tree is a reference to the note, while the name of the reference is itself a reference to the commit, from the data model, that is being annotated.
- The **commit** objects aggregate the history of all the notes of the repository.

Figure 3.11 illustrates graphically how the git notes model is used by example.

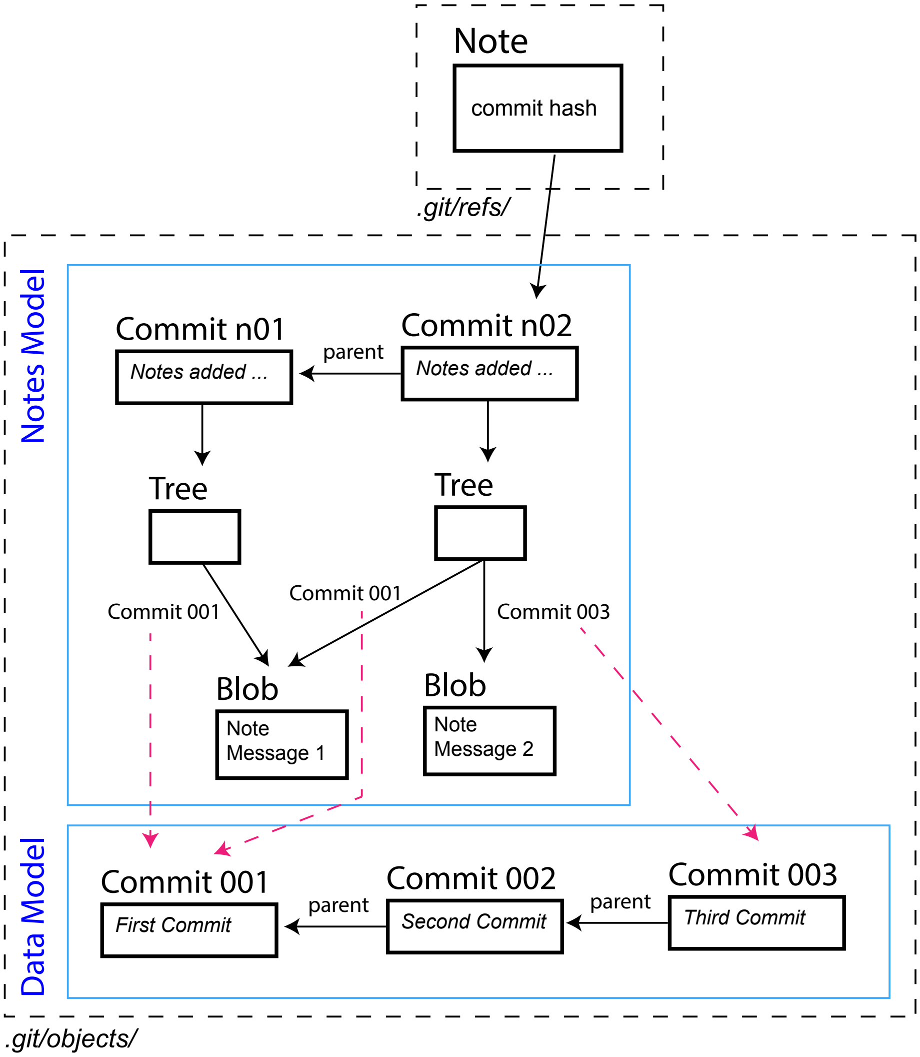


Figure 3.11: Example of git-notes.

3.3.3.4 Mercurial

Mercurial is an open source distributed VCS that shares the same scope and goals of git, but has a different data model design [93]. Git works by storing the state of the repository at each commit, while mercurial versions content incrementally, by storing data changes (*deltas*). The implications of the mercurial approach are discussed below.

The Mercurial Data Model

The mercurial data model is based on three types of first-class entities:

- The **file** is equivalent to the file abstraction used in file systems. A file also contains a reference to the *changeset* it belongs to, so that it is possible to know to what particular point in time of the repository this belongs to.

- The **manifest** defines the state of a repository at a given time. The manifest contains references to all the files of the repository at a given snapshot, their permissions, and their logical locations.

- The **changeset** records the changes of the repository in relation to the previous changeset.

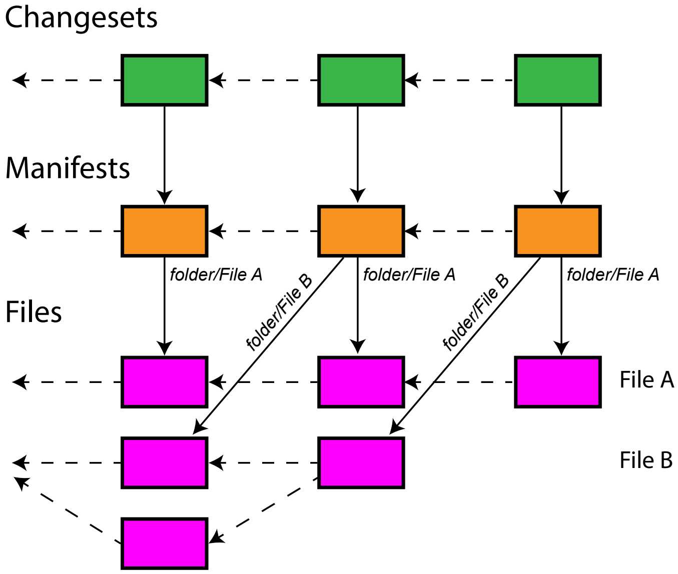


Figure 3.12: Mercurial data model.

Each entity also contains a reference to its previous versions, if any. The data structures used by mercurial are described in the next subsection.

Mercurial Internals

In mercurial, each entity — whether of type file, manifest, or changeset — is recorded within a *revlog*. The **revlog** is an index data structure of fixed-size records for each managed entity. There are three types of *revlogs*:

- The **file revlog**. There is one file *revlog* per file, with each *revlog* storing all the versions of a file. Each version is identified by a **nodeid**, which is the hash of the content of the file and the two parents it originates from.[80] Small files are stored directly in the file *revlog*, while large files are stored separately.
- The **manifest revlog**. This defines the state of a project at a given time by listing each file and its nodeid, which identifies the version of the file within its revlog (see Snippet 3.1)
- The **changeset revlog**. This is an index of all the files changed at a particular commit, plus the date of commit and a reference to the committer (see Snippet 3.2).

The most important difference between the mercurial and git model is that mercurial stores only the differences, or deltas, between files (*file revlog*) and repositories (*changeset* and *manifest revlogs*). Therefore, retrieving a particular version of a file involves getting the original file and then applying the appropriate deltas. As the history of a file grows, reading becomes slower. As an optimisation step, mercurial occasionally stores the entire content of a version, reducing the number of deltas to apply.

```
$ hg --debug manifest # outputs the current manifest
3f1854bbbc477130128ba81df0637a1ee1a96083 644 README
c6f503126dbe9f7d787b030fa8428dc36bce4b3c 644 documentation/CS4099___Final_Report.pdf
0bbac2bc381c2a7d951015c636ab6da09e99fe1b 644 documentation/CS4099___Mid_report.pdf
a17f8854f2994272e09ecde6ebf98d285ab7eace 664 documentation/feedback.html
c3ce0309360dfa5722f8efe2f44bd09a0367f433 624 documentation/feedback_files/css
...
```

Snippet 3.1: Example of a mercurial manifest. For each file, the manifest stores its nodeid, permissions, and logical path within the repository.

```
hg debugdata .hg/store/00changelog.i 165 # 165 is the changeset (or changelog) number
d619d180b24b4dbb9f25ea56e00d59f088f51548 # the corresponding manifest id
sic2 # the committer
1492699213 -3600 # the date since the epoch plus the offset from UTC, both expressed in seconds
```

[80] `nodeid = sha1(min(p1, p2) + max(p1, p2) + contents)`. p1 and p2 are the parents' ids for the node. If the parent id is null, then the id used for hashing takes the format: `0000000...000`

```
README # the list of changed files, followed by the commit message
documentation/CS4099___Final_Report.pdf
...

Added the final report # commit message
```

Snippet 3.2: Example of a mercurial changeset

3.4 Cloud Storage

3.4.1 Infrastructure as a Service Storage

Cloud services can provide low-level storage abstraction — IaaS storage — of four types: block storage, object storage, file storage, and database service. Each of these services is related to a different type of data abstraction. Block storage services provide storage at the block level over which users can configure file systems, databases, or their applications. File storage is usually provided through files systems like GFS and HDFS (see Section 3.2), while database services are usually provided by building easy-to-use interfaces on top of existing database technologies (see Section 2.2.2). This section discusses the data model and architecture of Amazon S3 [20]. Other IaaS storage solutions, such as Backblaze's B2 Cloud Storage [125] and DigitalOcean Spaces [120], provide similar abstractions and functionalities.

3.4.1.1 Amazon S3

Amazon S3 (Simple Storage Service) is an object storage cloud service provided by Amazon, that provides high scalability, high reliability, and low failure rate. The next subsections describe in detail the storage metaphors it uses, how access control is regulated, and its architecture.

Buckets and Objects

Data in Amazon S3 is stored as objects, which are organised in buckets. An **object** consists of data[81] and metadata and it is identified by its name. A **bucket** is a named

[81] Objects can be of size up to 5 TB.

container that can store a potentially unlimited number of objects and is regulated by a set of admin-defined rules. Amazon S3 is accessible to developers via SOAP, REST API, or using one of the provided AWS SDKs.[82] The name of an **object** must be unique within a bucket. Objects are accessible through URL addresses with the following scheme:

```
http(s)://<bucket_name>.s3.amazonaws.com/<object_name>
```

As of today, Amazon S3 stores trillions of objects and handles millions of requests per second [148].

Versioning

Object versioning is optional and must be enabled at the bucket level. All versions are assigned a version id, which changes as data is updated, while the name of the object remains unchanged. Amazon S3 supports only linear versioning (*i.e.*, no branching) and object deletion results in the creation of a special deleted version object (see Figure 3.13). Versions are accessed specifying the bucket name, the object name and the version id:

```
http(s)://<bucket_name>.s3.amazonaws.com/<object_name>?versionId=<version_id>
```

Metadata and Tags

An object's metadata is a set of key-value pairs (attributes). Some of the default metadata attributes are the data length in bytes, the data of last modification, and the content-type of the data. Custom metadata attributes for an object can also be set. When versioning is enabled, metadata is assigned to a particular object's version, so that a change in the metadata results in a new object's version.

In addition, an object can be associated with one or more mutable tags (up to ten), which are key-value pairs that can be used to group together objects stored within the same bucket. While metadata attributes are used to describe objects, tags can be used to enforce control over groups of objects, as described below.

[82] Amazon S3 integrates the BitTorrent protocol too, but only to retrieve data from S3 to a BitTorrent network.

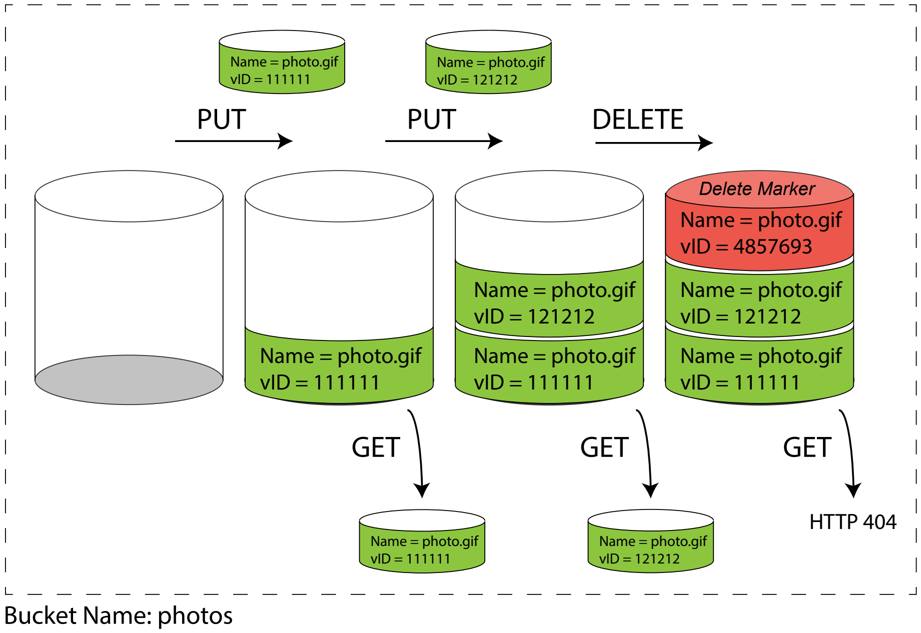


Figure 3.13: Amazon S3 versioning diagram, derived from [149]. The *vID* is the version id of an object's version.

Access Control and Data Protection

Amazon S3 enforces access control both at the resource (*i.e.*, objects and buckets) level, via bucket policies and ACLs, and at the user level, through the AWS IAM (Identity and Access Management) service [150, 151].

Furthermore, Amazon S3 protects data both in transit and at rest via encryption [152]. Data encryption is enforced at a per-object granularity using either a client master-key or an Amazon-generated master-key.

Policies and Notifications

Amazon S3 enables users to define policies for the buckets they administer. Policies can be enforced on specific objects or set of objects that match some given tags or whose names start with a given prefix. The following are examples of policies:

- Define the life cycle of an object, so that after an interval of time from its creation the object is automatically migrated to cheaper and slower-access storage services (Amazon S3 Standard IA[83] and Amazon Glacier [153]).

[83] IA stands for Infrequent Access.

- Define a policy such that all new objects are replicated to another bucket.
- Define a policy such that all objects matching a given tag are replicated to a set of selected buckets.

Furthermore, it is possible to enable notifications on a bucket, so that whenever an event occurs (*e.g.*, an object is added, an object is deleted, etc.) other AWS related services can be notified and act upon it.

Architecture and Performance Enhancements

Amazon S3 is implemented as a distributed storage system with per-bucket replication. Amazon S3 storage is distributed across multiple global data centres. The internal details of its architecture, however, are unknown due to the closed nature of Amazon.

Amazon S3 provides also a set of mechanisms to optimise the performance of an application using it. Some of these mechanisms are the following:

- **Multipart Uploads**. Objects can be divided in chunks that can be uploaded in parallel and re-assembled once they are all stored on Amazon S3.
- **Range-based downloads**. Objects can be downloaded in smaller chunks, enabling parallel downloads.
- **Cross Region Replication**. Objects can be replicated across multiple continents so that latency is reduced on object retrieval.

3.4.2 Software as a Service Storage

SaaS storage provides easy-to-use cloud storage for the average end-user. In particular, SaaS storage allows users to backup the files stored in a particular folder of their computer over the service's servers. Users, in addition, are able to use SaaS storage applications from multiple machines, which are automatically synchronised. Currently, all the main SaaS storage solutions are owned by companies that share only partial information about their storage models and architecture design. Some SaaS storage solutions are: Dropbox [18], OneDrive [19], Google Drive [124], BlackBlaze [125], Amazon Drive [154], Apple iCloud [155], and Adobe Creative Cloud [156].

In the remaining part of this section, the Dropbox solution is described in detail, since its design and architecture is well documented. A brief summary of the Google Drive, OneDrive, and Adobe Creative Cloud solutions follows.

3.4.2.1 Dropbox

Dropbox is a storage solution that allows end-users to backup the files stored in a particular folder of their computer to Dropbox's servers. Users can interact with Dropbox directly via a browser or through a Dropbox client installed on one or multiple machines, which are automatically synchronised with eventual consistency.

Files, Metadata, and Versions

In Dropbox, files are represented differently on the client- and the server-side. On the client-side, files are represented using the same abstractions as the local file system. On the server-side, files are represented as collections of immutable blocks of up to 4MB. Each file and its blocks are uniquely identified through their hashes (SHA-256), used to provide integrity of the stored data and block-level de-duplication. End-users can also set custom metadata for a given file, as explained in [157].

All files associated with a given user are said to belong to a specific **namespace**. Files stored within a namespace are isolated, and thus not de-duplicated, with the files of other namespaces. In doing this, two identical files that have the same hashed identifier are stored twice on Dropbox server if belonging to different namespaces. This mechanism is used to prevent attacks where one can learn whether a particular file is stored in Dropbox and then mimic having the file, given the known hash, in order to gain access to it. This attack was shown for the first time in 2013 and is known as a dropship attack [158]. Shared folders represent namespaces over which multiple users have access to.

Furthermore, Dropbox retains previous versions of a file for a limited amount of time [159], which are accessible to users via the Dropbox web interface. The actual versioning model is unknown.

Architecture

The known architecture of Dropbox (early 2012) consists of three important components: the client-side, the storage servers, and all the services that control and store metadata information about users, data, and notifications [160, 161] (see Figure 3.14).

This clear separation of services has enabled Dropbox to scale to hundreds of millions of users. When data is uploaded/downloaded to/from Dropbox, the client interacts with the load balancer to establish a connection with the block server, which is then responsible for controlling the rest of the data transfers. Drago *et al.* [160] have measured that over 90% of the Dropbox traffic (in terms of throughput) is due to data being uploaded/downloaded, which is handled by the storage services while keeping the interactions with the metadata services to a minimum. Figure 3.14 shows Amazon S3 as being the IaaS storage behind Dropbox, but most of the storage has been migrated [162] to *Magic Pocket*, Dropbox's own IaaS storage infrastructure, over the last few years. All data stored on Dropbox is protected using AES 256-bit encryption. Moreover, uploaded files are processed through a metadata server, which stores all the relevant metadata information about the file on database [163], while the notification server is informed of any changes within the namespace and notifies all the clients that have access to it.

A user making changes to a file, or directory, from two or more client machines can result in a conflict state. Dropbox attempts to resolve the conflicts without any human intervention when synchronising the clients with the Dropbox service. When Dropbox is unable to resolve the conflicts, the conflicting versions of the file are stored on the client side. The user then has to resolve the conflicts manually, by deleting all the unwanted versions of the file.

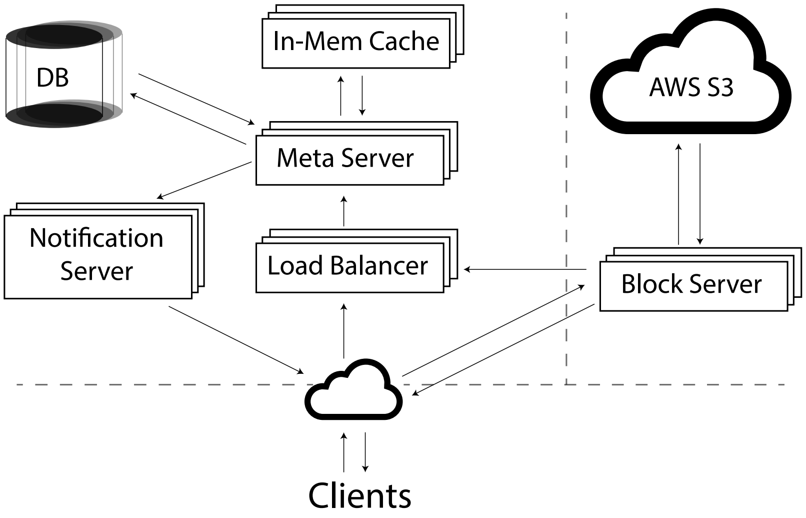


Figure 3.14: Dropbox architecture (early 2012) serving 50 million users. The arrows indicate the main direction of the requests made among the components involved. Diagram derived from the video at [161, minute 23].

Magic Pocket

Magic pocket is an immutable block storage system that Dropbox, Inc. has developed to substitute Amazon S3 over the years. Magic pocket is designed to provide secure and highly available storage [164].

Magic pocket stores files as a collection of **blocks** of size up to 4MB. All blocks are compressed, encrypted, and assigned to a unique key, such as a SHA-256 hash of the block. Blocks are aggregated together in **buckets** (1GB in size maximum) in order to improve IO performance when large amounts of data is moved/copied between Dropbox storage servers. In addition, blocks that are uploaded around the same time are stored closely together, thus exploiting temporal locality.

LAN Sync

The main performance bottleneck of SaaS storage is the bandwidth limit between the client and the service's servers. Dropbox attempts to overcome this bottleneck via LAN Sync [165]. LAN Sync is a feature built in Dropbox clients that can be optionally enabled to allow data exchange directly between clients that reside in the same network. Each client periodically broadcasts its presence over UDP (on port 17500) to other machines in the network, which are always listening. Whenever a file has to be downloaded, the client asks all other known nodes in the local network if they have it, otherwise the file is downloaded directly from Dropbox. LAN Sync operates on data files only and within the scope of the client's namespace. Metadata, instead, is always synchronised with the Dropbox servers, so that it is easier to enforce consistency across multiple clients.

The Placeholder Metaphor

Smart Sync (previously known as Project Infinite) is a feature of Dropbox that allows users to handle very large collections of data using a limited amount of hard disk space by storing only references — **placeholders** — of files and folders, which are retrieved on-demand. Unfortunately, Smart Sync is available only to Dropbox business users and its internal structure and format could not be studied.

3.4.2.2 Google Drive and OneDrive

Google Drive is a SaaS storage solution developed by Google LLC [124]. The client-side data model provided by Google Drive is based on files and folders, exactly like Dropbox. Moreover, the data managed by the Google Apps (*e.g.*, Google Docs or Google Photos) can be automatically stored to Google Drive. Google Drive also supports the placeholder metaphor [166].

OneDrive is a SaaS storage solution provided by Microsoft Corp. [19], similar to both Dropbox and Google Drive. Lindley *et al.* [167] have recently proposed an alternative metaphor to the file, called the *file biography*, in which a file is represented as an entity that changes over time through versions and can exist in multiple locations, through actions like sharing, copying, licensing, and cloning, while retaining its unique identity. In the examples presented in the paper, data stored under the file biography metaphor is integrated with OneDrive and Microsoft Word, so that its new semantic operations are preserved as data is synchronised to/from the cloud.

The server-side data models for Google Drive and OneDrive are unknown.

3.4.2.3 Adobe Creative Cloud

Adobe Creative Cloud (CC) is a collection of software applications and cloud services, developed by Adobe Systems, Inc., for creative professionals who work with digital photography, videos, and graphics in general [156]. CC provides a similar set of services to the other SaaS storage solutions presented above. The file metaphor used by the CC mobile clients, however, differs from the standard file representation used by other SaaS storage solutions. Goldman *et al.* have proposed and implemented DCX (Digital Composite Technology), a manifest-based data format that aggregates multiple independent components of a file together [168]. For example, a Word or PDF document can be represented in DCX as a collection of multiple components, containing the text, the formatting, the provenance, or the images. Using the DCX abstraction it is possible to have (I) semantic de-duplication, where semantically meaningful components of the file (*e.g.*, images, text, and metadata information of a PDF file) are de-duplicated rather than blocks of fixed size; (II) different data views based on context; and (III) the ability to record the provenance of the data independently of the data itself. (I) enables better distribution and synchronisation by avoiding files being fragmented in blocks, which are semantically unrelated. (II) results in better network usage and user experience for end-users, but it also means that a file can be perceived and used differently depending on context. Provenance, as in (III), can help end-users to understand better content and how, where, and when it originated. Further, DCX files can be embedded within other DCX files, so that it is possible to create even richer data abstractions.

The implementation of the DCX format can also be used through other SaaS storage solutions as well.[84]

[84] Adobe Systems, Inc. has released an implementation of the DCX format for Dropbox under Apache License 2.0. `https://github.com/Adobe-Digital-Composites/Digital-Composites-ObjC` [last accessed on 29/03/2018].

3.4.3 Multi-Cloud Storage

Multi-Cloud storage, also known as the *Cloud of Clouds* approach, is an architectural design that consists of combining multiple independent cloud storage services into one [26]. Many solutions have been proposed recently [169, 170, 171, 172, 173, 29, 174, 175], some with a particular focus for resource-constrained devices [28, 176, 177].

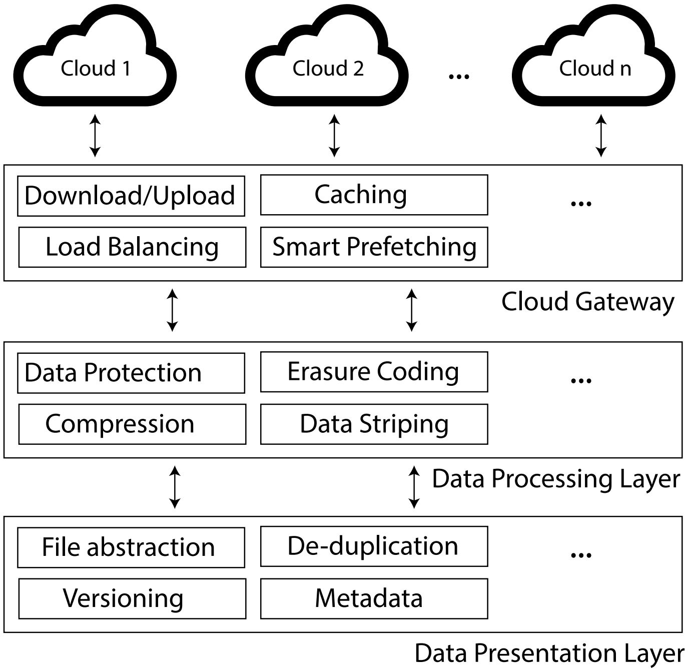


Figure 3.15: Example of a multi-cloud storage architecture. Diagram derived from [28]

A multi-cloud storage service is implemented by providing a storage interface that can be mapped across all the supporting services, as shown in the diagram at Figure 3.15. Data added to a multi-cloud storage service is then replicated, as a whole or after being erasure coded, to the actual services. The independence of the used services results in a system that has: higher availability, better security (especially if data is chunked first and then replicated), no vendor lock-in, and better resistance to services failing. Multi-cloud storage presents some disadvantages too. Firstly, multi-cloud storage services can provide only limited capabilities, defined by the least common denominator operations among the supporting services. Secondly, the development and maintenance of multi-cloud services are complex and expensive due to having to manage services that provide different workflows and APIs. Finally, multi-cloud storage services still may include a central infrastructure layer to control the existing underlying services and that can be a single point of failure for the system.

3.5 P2P

3.5.1 Overlay Networks

P2P storage systems are a type of shared-nothing distributed storage system (see Section 2.2.4.3), where nodes, or peers, are all treated as equals and can act both as clients and servers. P2P networks implement virtual overlay networks on top of the physical network. The type of overlay network defines the behaviour and performance of the P2P network. This section summarises briefly the types of overlay networks.

3.5.1.1 Unstructured Overlay Networks

In unstructured overlay networks no well-defined relationships between peers or between peers and the data they store exist [178]. Early P2P systems, such as Gnutella [179] or Freenet [180], are based on an unstructured overlay network. In such systems, peers usually interact with their neighbours only, such as when peers join or leave — knowingly or due to failures — the network (this is known as **churn**) or when data is being looked up. However, locating content in an unstructured P2P system can be hard. Looking up for data among neighbour peers can be inefficient and not even return a result, especially for networks of hundreds, thousands, or even more peers. Heuristic routing strategies have been devised to locate content in the network, such as flooding, used by Gnutella [179], or hill climbing, used by Freenet [180]. Understanding these strategies, however, is not the focus of this thesis.

3.5.1.2 Structured Overlay Networks

Structured overlay networks are defined by a set of rules, such that peers are arranged in structures like rings or trees [178]. In structured overlay networks, data is placed deterministically over peers of the network. In a structured overlay network, every node is assigned a unique identifier from a large identifier space (*e.g.*, SHA-1 identifiers belong to an identifier space of 2^{160} entries). Each node is responsible for a set of identifiers that is closest to its own.[85] Data is also assigned an identifier (*e.g.*, the hash of the data) and stored in the node that manages the identifier space that contains such identifier. Examples of structured overlay networks are Chord [181], CAN (Content-Addressable Network) [182], Pastry [183], Tapestry [184], and Kademlia [185].

Structured overlay networks manage content through a **distributed hash table** (DHT), a decentralised data structure that provides a content storage and lookup service [186, 178]. A DHT provides the following two interfaces:

$$put(data, data\_key)$$

$$get(data\_key) \rightarrow data$$

Structured overlay networks are more efficient than unstructured overlay networks when looking up exact data matches, but perform worst when searching content by its attributes or range-based queries, since node and data identifiers do not reflect the actual nature of the data.

3.5.2 P2P Storage Systems

A large number of P2P storage systems have been developed over the years, both from the academic [112, 187, 188, 189] and the open source communities [113, 190, 191, 57, 192]. This section discusses in detail only the data models and architecture of OceanStore [112], IPFS [113], Perkeep [190], and Tahoe-LAFS [191, 193]. These systems have been chosen because of their historical significance, interest by the scientific and open source communities, and their properties and data models. However, many P2P storage systems share ideas and design choices with the ones presented below.

[85] The definition of identifier distance depends on the actual overlay network implementation.

3.5.2.1 OceanStore and Pond

OceanStore is the design of a P2P global-scale persistent storage system relying on the Tapestry overlay network [184] and proposed by Kubiatowicz *et al.* [112] in the year 2000. Pond is a prototype of a P2P storage system largely based on the OceanStore design and released under the BSD license [194].

Data Model

The OceanStore data model [112, 194, 195, 196] is based on data objects which are versionable, with each version identified by a GUID, calculated by hashing the content of the versioned data (VGUID). A data object can either represent a file or a directory. The collection of all the versions of an object is identified by the active GUID (AGUID), generated by hashing the owner's public key and the human-readable name of the object. Mutability is achieved by aggregating the different versions of the same objects through the same AGUID. Each version object is subdivided into immutable blocks (8KB by default in the Pond implementation), each identified by a GUID, generated by hashing the block data (BGUID) (see Figure 3.16). Metadata, such as the owner's id or user specified data, is stored within the version entity of an object. The overall structure is a Merkle tree.

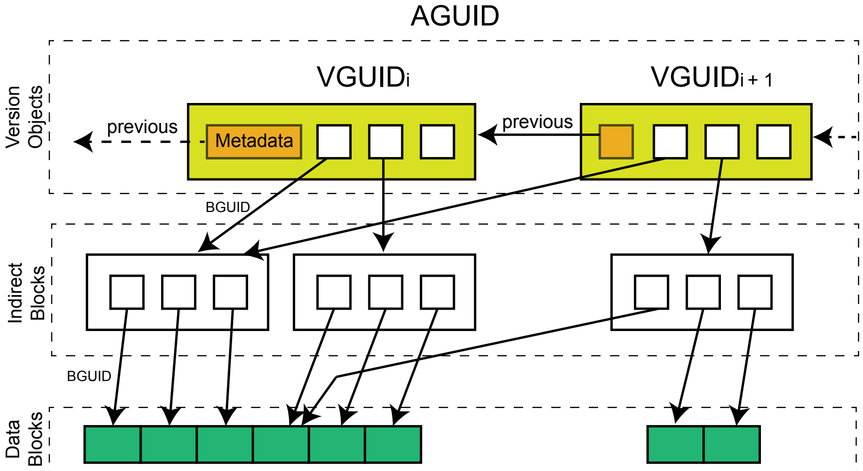


Figure 3.16: Diagram of the OceanStore data model. Each version is a Merkle tree of read-only blocks. Versions are identified by the VGUID, while all blocks are identified by BGUIDs. The data of a version is stored in the leaf nodes of the B-tree (green boxes). The group of all versions is identified by the AGUID. Diagram derived from [194].

Architecture

OceanStore is built on top of a hybrid overlay network (*i.e.*, a network where some nodes are organised in a structured overlay network and others are not), with nodes identified by ids generated by hashing their public keys and locations (*e.g.*, ip or hostname), as suggested by Mazières [197]. The immutable components of the data model, the blocks and the object versions, are managed over a Tapestry network. Update operations (*i.e.*, mapping the active GUID of an object to a specific version GUID) are managed through primary-copy replication, where each uploaded object is assigned to a **primary replica**, a small set of nodes that controls data updates — using a Byzantine agreement protocol to agree on the updates — and enforces access control restrictions, via ACL. Further, objects are stored to two other types of nodes: **archival storage peers** and **secondary replica peers**. The diagram in Figure 3.17 shows the different types of nodes in OceanStore. After an object is updated through the primary replica, erasure coding is applied at the block level and distributed across the Tapestry network to the archival storage peers.

Data objects are also replicated to secondary replica peers, which are used to increase data availability and bring data objects closer to other clients.

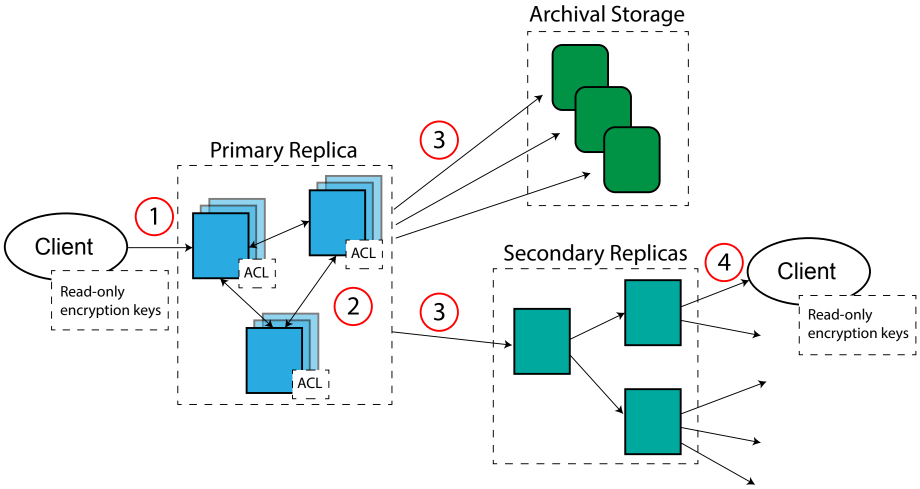


Figure 3.17: Diagram illustrating the process of updating a data object in OceanStore. Diagram derived from [194]. First of all the client sends the update object to the primary replica (1). The primary replica nodes agree on the update (2) and then propagate the object to the archival storage nodes and the secondary replica nodes (3). Other clients can retrieve the object from the secondary replicas(4).

Access Control

The OceanStore model enforces access control over data via the primary replica [112]. A user can be granted read and/or write permission on the data. Read-only permission is achieved by encrypting the data with an encryption key, which is distributed to all the users to whom read access should be granted. A user is granted write permission by adding his/her public key to an ACL (see Section 2.1.8.1), which is securely managed by the primary replica. Data writing operations must be signed and verified against the ACL to be accepted.

3.5.2.2 InterPlanetary File System

IPFS (InterPlanetary File System) is an open source (MIT) P2P file system with versioning built-in [113]. IPFS consists of a distributed data model and a P2P architecture. The IPFS data model is constructed on top of two other projects — developed by the same open source community — called IPLD and Multiformats. These auxiliary projects are described briefly below.

IPLD

IPLD (InterPlanetary Linked Data) is the generic data model framework upon which the IPFS data model is built. IPLD models data as content-addressable and allows data instances to be part of a unified data model [198]. The IPLD open source community has developed IPLD-based models for IPFS [113], git [92], Bitcoin [199], Ethereum [200], ZCash [201], and Torrent [202]. The IPLD data structures can be expressed in multiple formats, such as JSON, YML, CBOR, or XML. The advantage of using IPLD is that it is possible to define first class entities using a consistent data structure, so that content can be easily referenced and exchanged among different actors. The IPLD model is based on the following three constructs: Merkle-links, Merkle-DAG, and Merkle-paths.

Merkle-Link

A Merkle-link defines a directed relationship between two objects and consists of a string key and the hash of the target object, which is stored within the source object.

```
{
   "/" : "QmKq5sNclPz7QV2+lfQIuc6R7oRu0="
}
```

JSON 3.3: Example of a Merkle-Link.

Merkle-DAG

Content-addressable objects which are linked via Merkle-links form a Merkle-DAG, a directed acyclic graph of objects that form a Merkle data structure. Section 3.3.3.2 described the Merkle data structures and their properties in more detail.

Merkle-Paths

A Merkle-path is a Unix-like path that allows Merkle-links to be aggregated together to address nested objects.

```
# Object with hash:  A00000EBiSfc7D25570fFCw9QPDFA=
{
   "foo" : {
      "bar" : {"/" : "QmKq5sNclPz7QV2+lfQIuc6R7oRu0=" },# This is a Merkle-Link
      "baz" : "QmKq5sNclPz7QV2+lfQIuc6R7oRu0=", # This is not a Merkle-link
      "fred" : "cat"
   }
}

# Object with hash:  QmKq5sNclPz7QV2+lfQIuc6R7oRu0=
{
   "id" : 123,
   "info" : {
      "name" : "Bob the Cat"
   }
}
```

JSON 3.4: Examples of linked Merkle objects.

The data structures in the example of JSON 3.4 can be traversed using the following paths:

- `A00000EBiSfc7D25570fFCw9QPDFA=/foo` - returns the foo object

- `A00000EBiSfc7D25570fFCw9QPDFA=/foo/bar` - returns the second object with hash `QmKq5sNclPz7QV2+lfQIuc6R7oRu0=`

- `A00000EBiSfc7D25570fFCw9QPDFA=/foo/fred` - returns the "*cat*" string

- `A00000EBiSfc7D25570fFCw9QPDFA=/foo/bar/id` - returns the id value *123* for the object with hash `QmKq5sNclPz7QV2+lfQIuc6R7oRu0=`

- `A00000EBiSfc7D25570fFCw9QPDFA=/foo/bar/info/name` - returns the string "*Bob the Cat*", stored as part of the object with hash `QmKq5sNclPz7QV2+lfQIuc6R7oRu0=`

Multiformats Project

The *multiformats project* defines a collection of protocols which are self-describing, in order to promote better interoperability, extensibility, and backward compatibility among applications and over time. In this thesis, only the *multihash* and *multiaddr* protocols are examined.

Multihash

Multihash is a self-describing protocol for hashes that is constructed through the following pattern:

```
<hash-func-type><digest-length><digest-value>
```

Where, the *hash-func-type* is a positive integer that defines the hash function (*e.g.*, 0x11 for SHA1, 0x12 SHA2-256, 0xb240 for BLAKE2b-512); the *digest-length* is the length in bytes of the hash value; and the *digest-value* is the actual hash value calculated using the specified hash function and of the specified length. Hash values are always of the same size for a given hash function, so the *digest-length* is needed simply for implementations that are unable to parse a given hash function.

The following is an example of a multihash for SHA1:

```
11148a173fd3e32c0fa78b90fe42d305f202244e2739
```

where the first two digits - 11 - indicate the function SHA1, the third and fourth - 14 - the length of the hash in hexadecimal and the rest of the digits are the hash value.

Multiaddr

Multiaddr is a self-describing protocol for network addresses that is constructed through the following pattern:

```
(/<addr-protocol-str-code>/<addr-value>)+
```

Where, the *addr-protocol-str-code* is a string identifying the network protocol (*e.g.*, tcp, udp, http, etc.); while the *addr-value* is the actual network address value. Multiaddresses can be concatenated together.

The following is an example of a multiaddr for an ip/tcp/http address:

```
/ip4/127.0.0.1/tcp/80/http/baz.jpg
```

Data Model

The IPFS data model, built on top of the IPLD model and the multiformats project, consists of four object types:

- The **blob** object, which stores data similarly to git.
- The **list**, which aggregates blobs and lists. A list is meant to be used to represent a large file composed of multiple de-duplicated components.
- The **tree**, which aggregates blobs, lists, and other trees. A tree is used to represent a directory.
- The **commit**, which links content together as it mutates over time.

Figure 3.18 illustrates the relationship between the entities defined by the IPFS model. The IPFS model is partially inspired by the git model, from which it borrows also the terms blob, tree, and commit, but it allows its entities to be distributed independently from each other, while in git all entities are enclosed within a repository.

Architecture

The IPFS architecture is based on a DHT, similar to the one used by the Self-certifying File System [197, 203], where nodes self-assign ids by hashing their public key. The node's

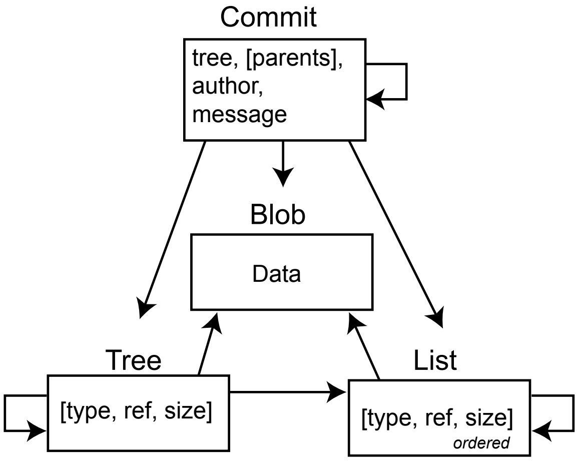


Figure 3.18: IPFS data model.

keys are also used to sign network requests. The id of the node is expressed using the multihash format, while the location of a node is expressed using the multiaddr format. IPFS uses a block exchange protocol, **BitSwap**, where nodes exchange blocks irrespective of the file they belong to.

3.5.2.3 Perkeep

Perkeep (previously Camlistore - *Content-Addressable Multi-Layer Indexed Storage*) is “a set of open source formats, protocols, and software for modelling, storing, searching, sharing and synchronising data” [190] (under the Apache-2.0 license) for personal data management. Perkeep is designed for personal storage, thus it is optimised to work with a small set of peers, unlike many other P2P storage systems.

Data Model

In Perkeep content is represented through the standard file and directory metaphors, modelled using a large variety of first-class objects. All objects are identifiable by their hashes. The diagram in Figure 3.19 shows the relationship among the supported object entities. The following are the main types of objects introduced by the Perkeep data model:

- The **blob** object stores data, similarly to the blob in IPFS and git. A blob is identified by a **blobref**, which is the hash of the blob.

- The **file/bytes** object consists of a collection of blobs. A file may contain metadata about the data stored in the blobs, while the bytes object does not.
- The **static-set** represents a collection of blobs or other static-sets, similar to the tree object in git.
- A **claim** is any object that is signed and contains a reference to the signer. Claims can be used to record arbitrary metadata, mark content as a directory, as deleted, or as shareable with other users.
- The **permanode** is a special type of claim object that enables Perkeep users to create static references to mutable objects. A claims has always a reference to a permanode and optionally to other Perkeep objects.

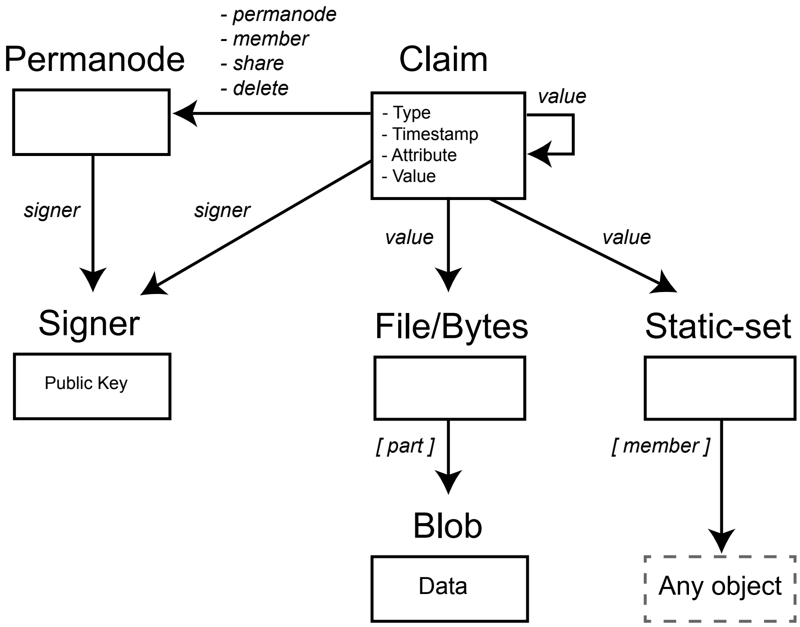


Figure 3.19: Perkeep data model.

Versioning

Unlike other P2P storage systems, such as OceanStore and IPFS, Perkeep does not rely on a Merkle data structure to manage versioning. Instead, content is versioned by creating new claim objects and their relationship, and ordering, is established using timestamps stored within the claim objects. The result of this versioning mechanism is that data is always versioned linearly (*i.e.*, branching and merging operations are not allowed).

3.5.2.4 Tahoe-LAFS

Tahoe-LAFS (Least-Authority File Store) is a decentralised cloud storage system designed to provide high privacy, integrity, and security over the stored content [56, 193].

Data Model and Access Control

Tahoe-LAFS uses the standard file and directory metaphors when presenting content to its end-users. However, it is also important to understand how data is stored and protected, since privacy, integrity, and security are the three main principles of the system.

Files in Tahoe-LAFS are either immutable or mutable and referenced through their *capability*. In Tahoe-LAFS, a capability is defined as a hash value which identifies the file and is generated differently based on the type of file:

- **Immutable file**. The capability is the hash of the file's data.
- **Mutable file**. This is a file that can change over time, while retaining the same capability [191]. A mutable file is associated with a public/private key pair and the capability is calculated using these keys. Two types of capabilities can be generated:
  - *Read-write capability*, represents the ability to get and update the file's data. The capability is the string `write-key:fingerprint`, where the write-key is the symmetric key used to encrypt the file's private key and the fingerprint is the hash of the file's public key.
  - *Read-only capability*, represents the ability to get the file's data. The capability is the string `read-key:fingerprint`, where the read-key is the hash of the write-key (*i.e.*, it can be generated only by someone who has write access to the file) and the fingerprint is the hash of the file's public key.

To avoid global file de-duplication attacks, such as *dropship* for Dropbox [158] (see also Section 3.4.2.1), a client can set a **convergence secret**, a random string never to be shared with other users and which is used when generating the hash of files. When using **convergence secret** two exact files will have different hashes and global file de-duplication attacks are avoided.

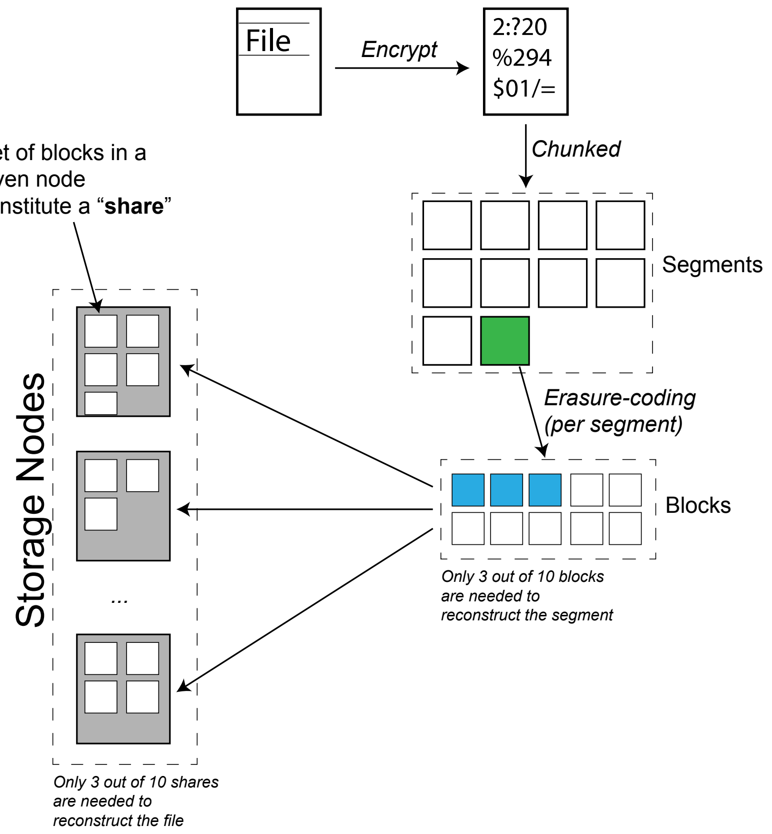


Figure 3.20: File addition workflow diagram for Tahoe-LAFS. A file is (1) encrypted; (2) chunked into segments; (3) segments are erasure-coded into blocks; and then (4) blocks are distributed across multiple storage nodes.

Finally, Tahoe-LAFS provides high levels of security and data availability by encrypting data, chunking it into multiple segments and applying erasure coding for each segment, as shown in Figure 3.20. The overhead cost of this process is traded-off with the ability of Tahoe-LAFS to reconstruct a file even when some of the redundant data is lost, corrupted, or temporarily unavailable. By default, erasure coding (see Section 2.1.6.2) is applied with (n, k) equal to (3, 7), such that only 3 blocks out of 10 are needed to reconstruct a segment. In this case, the storage space cost of erasure coding is higher than replicating content three times, but the system can tolerate a higher number of failures, in terms of node failures, data loss, and data corruption.

3.6 Context-Aware Storage

Organising and managing data in a storage system is challenging, in particular as the amount of data and the number of devices and services increases. The task of data management, such as replicating data for backup, can be achieved manually or via applications such as Time Machine or SaaS storage solutions. Another alternative, which has been explored over the last decades, is context-aware storage.

One of the most widely accepted definitions of context is the one by Dey and Abowd [204, 205]:

> *Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.*

The problem of how data should be organised based on context has been addressed multiple times over the years. The Semantic File System [206], one of the first context-aware storage solutions, allows users to define rules to extract metadata information about the stored files and uses it to provide dynamic views of the data. Freeman and Gelernter proposed Lifestreams, a storage system where files are organised as a stream of ordered content along the time axis [207]. In Lifestreams, files are not necessarily associated with a name and their location is simply their relation to the time stream. Lifestreams classifies files based on their metadata, similarly to the Semantic File System, so that it is possible to resolve queries as such “all emails I haven’t responded to” or “all the pdf files accessed yesterday”. Lifestreams, however, does not scale well when the number of files increases. The system is unable to extract metadata for all files, while users would be overwhelmed by the amount of content presented.

An alternative approach to model context storage is proposed by Román *et al.* with Gaia, a ubiquitous operating system with a built-in context-aware storage: the *context file* *system* [208]. In Gaia, data is organised through dynamic data views that have knowledge of the current environment context (*e.g.*, number of people in the room, device used, *etc.*). WinFS was an attempt by Microsoft to build a storage system for the Windows operating system which provided dynamic views of the stored data [76], similarly to the Semantic File System, Lifestreams, or Gaia. WinFS, however, also had the ability to define data types and relationship between files. For example, documents could be associated with metadata of type 'contact', where each context could be related to a picture that identifies it. Defining these types of relationships allowed users to express rich search queries, such as "find all the image files that have been taken in Canada by the calendar contact user X, modified by user Y yesterday".

Veeraraghavan *et al.* proposed the concept of quFile, where a file is associated with multiple views, each returned to the user based on particular parameters of the current context (*i.e.*, the device used, the bandwidth of the network, *etc.*) [209]. The file biography [167] and Adobe's DCX [168], presented in Sections 3.4.2.2 and 3.4.2.3, are other examples of context-aware data metaphors.

Other solutions of context-aware storage supporting entity-relationships have also been built over the years [210, 211, 212, 213, 214].

Advanced context-aware storage systems, like the Semantic File System, Gaia, and WinFS, have proved to be overly complex, hard to scale, or difficult to integrate with existing storage systems. Nonetheless, some of the basic ideas of context-aware storage are now applied over existing and widely used storage solutions. Automatic data management based on context is a concept used in other types of storage systems too. For example, Amazon S3's policies can be used to migrate the contents of a bucket to cheaper storage solutions as it becomes old and infrequently accessed, all without any human intervention.

The next subsections describe in more detail the Semantic File System and the quFile metaphor.

3.6.1 The Semantic File System

The seminal work by Gifford *et al.*, the semantic file system (SFS), is one of the first to explore how data storage could be enhanced by the automatic extraction of data attributes and tag-association [206]. The SFS automatically extracts metadata attributes via transducers. A **transducer** is "a filter that takes as input the contents of a file, and outputs the file's entities and their corresponding attributes" [206], thus acting as a classifier for the file when this is created or modified. Examples of transducers reported in the paper can extract meta information from mails, TEXfiles, compiled C objects, CAD projects, and C/Pascal/Scheme source code. The SFS provides dynamic views of the file system through virtual directories. A **virtual directory** is a directory whose name is interpreted as a query against the index of all files that have been classified by the transducers. For example, the path `sfs/owner:/simone/ext:/pdf` identified the virtual directory with all *pdf* files owned by the user *simone*.

3.6.2 The quFile

The quFile is a file metaphor presented by Veeraraghavan *et al.* that enables multiple physical representations of a file to be grouped under the same logical file view [209]. For example, a quFile for a photo can be represented by multiple files of different formats (*e.g.*, jpg, png, tiff) and resolutions (*e.g.*, 640x480, 1024x768, 1920x1440), but only one of these representations is returned to the user based on the context, which could depend on the type of device or the bandwidth of the network. Other storage systems provide context-based views of the data too, based on the screen size [215, 216] or the battery status of the device [217]. Similarly, the MPEG file format, in 2011, has been extended to provide dynamic adaptive streaming over HTTP (MPEG-DASH) so that the quality of a video depends on the quality of the network [218]. Adobe's DCX format can also be used to provide context-aware storage by returning different views of a file based on the context [168].

The views and operations provided by quFiles are programmable and are described by four types of policies:

- **Name policies** allow quFiles to be mapped to zero or more named logical representations of the data.
- **Content policies** allow quFiles to return different content based on context. The content returned can be stored in the file system or created dynamically, where possible. An example of a content policy is: “return JPEG if cached, else return PNG”.
- **Edit policies** specify what operations are allowed on the content. There are three types of edit policies upon content modification: allow, disallow, create a new version.
- **Cache policies** allow content to be cached for faster data retrieval. These types of policies are useful when using quFiles in a distributed system, where data is not always available and caching can have a significant impact on the overall storage system.

3.7 Conclusions

This chapter has presented a general overview of the state of the art about data management systems (DMS). This chapter addresses different types of DMS, from local file systems to P2P storage, and for each of them it describes and discusses their data abstraction, access control models, security mechanisms, metadata management, and architecture. Given the literature review presented above and the systems taken into account, the following statements can be made:

- The file and directory are the two most fundamental abstractions in storage systems. These abstractions were introduced decades ago and the way they are used today differs from what they were originally meant for. However, files and directories are still relevant nowadays, since end-users are familiar with them.

- The ability of versioning content is becoming a common property of many storage systems. Versioning can be achieved in multiple ways, each with its own advantages and disadvantages.

- Metadata can vary from system to system, but its purpose is central to many storage solutions. The ability to model metadata correctly and position the metadata management component of a system properly into the architecture is key to achieve reliability, consistency, and scalability.

- Context awareness is key to provide richer experience to end-users and enhance the common file/directory abstractions.

In particular, the following can be stated for each storage system class and how it relates to the hypotheses stated in Chapter 1.

- **File systems** are built around the file and directory abstractions. These abstractions, however, are not designed to be easily described by metadata, versioned, and distributed. Moreover, file systems do not provide abstractions to describe how content should be managed over time and over a set of distributed nodes.

- **Networked file systems** are based on the same abstractions as file systems, but provide a set of protocol that allow content to be managed over the network. Nonetheless, versioning is not part of their data models and/or protocols and granular automated control of the data is usually provided at the application level only.

- **Traditional versioning systems** do provide the ability to record how content changes over time and, sometimes, what metadata describes a particular version. However, these systems are designed to work as self-contained systems, rather than distributed systems. For example, backup solutions are designed to version data for a particular device only, while version control systems, such as git or mercurial, do version content within the scope of a repository, rather than a global collection of data. Moreover, versioning systems do not provide control mechanism to define how data should be replicated and protected.

- **Cloud storage solutions** are of two main categories: infrastructure as a service storage and software as a service storage. The former solutions are often built around an object-based data model, with no data aggregation and versioning. The latter solutions, instead, provide users a data model that resembles the one used by traditional file systems. Cloud storage vendors usually provide versioning and metadata capabilities, but these are built as part of their infrastructure rather than the actual data models in use. In addition, cloud storage solutions provide access-control and granular control over data, but these are centralised services not designed to work across multiple services. Multi-cloud storage solutions attempt to solve cross-cloud integration at the cost of providing the lowest common denominator data model among the solutions supported.

- **P2P** storage solutions can provide location transparency and access control over globally distributed data. Some solutions are also able to provide versioning over content. However, none models metadata as a separate versionable entity from data and allows granular automatic control over data. Finally, with the exception of IPFS, P2P data abstractions are tightly bound to a particular architecture design and data model implementation, thus making it harder to manage data across multiple different systems.

- **Context-aware storage systems** provide granular control over data, but their context models do not always take into account that data can be distributed across multiple nodes of a network and that data can change over time and thus be versioned. Moreover, these solutions are rarely built on top of abstractions that are independent of their implementations.

The systems presented in this chapter are also evaluated comparatively in Chapter 7 against the solution presented in this thesis.

Design Requirements

The objective of this work is to propose the design of a generic data model that satisfies the hypotheses provided in Chapter 1. This section outlines the requirements to design such a model and build a prototype of it.

The *end-user requirements* specify what features and behaviours the SOS should provide to the end-users.

The *model requirements* specify what the SOS data model needs to provide to satisfy the hypotheses **H1** and **H2** as stated in the *Introduction* Chapter (see section 1.3).

The *architecture requirements* consist of the functionalities needed for the SOS to work in a distributed environment.

4.1 End-User Requirements

EU-1 Location abstraction: data should be accessible irrespective of the location where it is stored.

EU-2 Accessibility abstraction: data should be accessible irrespective of the location it is accessed from.

EU-3 Users and Roles: a user (*e.g.*, Simone, Al, Graham) should be associated to multiple roles (*e.g.*, home, work, hobby) and have a different view of the SOS based on its current role.

EU-4 Data protection: users should be able to enforce arbitrary levels of data protections, in terms of what to protect and what encryption algorithms to use.

EU-5 Versioning: it should be possible to version data and metadata at different granularities.

EU-6 Automatic data management: users should be able to define rules to automatically manage data in a distributed environment. Rules define:

- Against what data the rule should be run (*i.e.*, what data should be automatically managed).
- Over what nodes the rules should be run.
- What actions should be applied on such data.

4.2 Model Requirements

M-1 Immutability: the components or entities that describe the model should be immutable.

M-2 Integrity: it should be possible to verify the integrity of the content managed through the model.

M-3 De-duplication: the model should support data de-duplication.

M-4 Self-describing entities: entities should be self-describable (see *Glossary of Terms*), so that no additional information is needed to use and process a given entity.

M-5 Independent entities: it should be possible to distribute content independently of any other modelled entity.

M-6 Computational work: it should be able to define computational work that runs over the stored content. It should be possible to distribute the computation across the system.

4.3 Architecture Requirements

A-1 CAP: the system should be designed to be always available and tolerant against network partitions (*i.e.*, AP compliant, see Section 2.1.5). Thus, data should always be available, assuming that it is reachable, and failures regarding some nodes of the system should be tolerated. This system will forfeit *strong consistency* across all available nodes.

A-2 Failure: the system should be able to cope with failures using an *eventually consistent model*.

A-3 Heterogeneity: the system should work independently of the type of hardware and operating system it runs on.

A-4 Horizontal scalability: the system should be able to scale horizontally (*i.e.*, adding nodes to the system).

A-5 Node configuration: it should be possible to configure a node in terms of what services it provides to the system.

A-6 Decentralisation: the system should be able to work without the need of a central authority.

A-7 RESTful API: a node should expose its services via a RESTful API.

The Sea of Stuff

This chapter presents the design of the *Sea of Stuff* (SOS), a generic model for distributed data storage that addresses the two hypotheses – **H1** and **H2** – presented in the *Introduction* Chapter, Section 1.3.

The design of the *Sea of Stuff* consists of a model and an architecture of nodes in a network. Section 5.1 presents how nodes, data, metadata, users, and computation are modelled. Section 5.2 discusses in detail the architecture of SOS nodes and the interaction between the different components of the model.

A prototype implementation of the SOS and example applications built on top of it are presented in the next chapter.

5.1 The Sea of Stuff Model

5.1.1 The Sea of Stuff GUID

The SOS is based on content-addressable entities that are identified by *globally unique identifiers* (GUID) (see Section 2.1.2). In the SOS, a GUID is defined as the value that results from a cryptographic hash function being applied over some content (the content that is being hashed depends on the type of the entity). The use of a cryptographic hash function allows both content-addressability and integrity verification (see Background Chapter) over content. Moreover, GUIDs, in the SOS, are formatted as to be self-describing. Such a property is achieved by formatting the GUIDs as following:

$$\langle\texttt{function name}\rangle\_\langle\texttt{base}\rangle\_\langle\texttt{hash value}\rangle$$

The *function name* is a string that identifies the hashing algorithm. For example, the following strings can be used as function names: MD5, SHA1, SHA256, BLAKE, BLAKE2. The *base* parameter defines the numerical base on which the hash is based (*e.g.*, base 16, base 32, or base 64). Finally, the *hash value* parameter is the actual hash. Examples of GUIDs derived using the hash function SHA256 and expressed in base 16 and 64 are the following:[86]

```
SHA256_16_d7a8fbb307d7809469ca9abcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592
```

```
SHA256_64_16j7swfXgJRpypq8sAguT41WUeRtPNt2LQLQvzfJ5ZI=
```

The terms GUID and reference will now be used interchangeably. To provide better readability, all the examples in the rest of this work will have GUIDs expressed as hexadecimal hashes of length five and the function name and the base parameter will be omitted (*e.g.*, d7a8a, 4e29a, 10304, ae510).

5.1.2 A Manifest-based Model

The SOS consists of content-addressable **entities** of different types described by manifests. A manifest is a self-describing and re-computable (see *Glossary of Terms*) structured 'document' that stores information about the entity it describes and that can be published in the SOS to allow the discoverability of the entity. Manifests are content-addressable by their hashes (*i.e.*, GUIDs). The GUID is derived by hashing the content related to the manifests, which depends on the type of manifest. Each SOS entity is described by a manifest of a particular type, which has a base structure as defined in Manifest 5.1.

```
{
  "type" : <Manifest Type>,          # Literal string, such as Atom, Compound, Version, etc.
  "guid" : <hash(Content to Hash)>, # The content to hash depends on the type of manifest.
  ...
}
```

Manifest 5.1: Base Manifest JSON structure.

[86] The string hashed is: "The quick brown fox jumps over the lazy dog".

5.1.3 The Node Model

In the SOS, data and metadata are stored in and retrieved from nodes. A node is any physical device in the SOS network that can interact with other nodes by sending or receiving information to or from them. A node serves as an access point to the SOS and is identifiable by a GUID. The node manifest (see Manifest 5.2) consists of the hostname and port number necessary to interact with it, a public key used to sign the requests made, and the list of services that it provides.

The pair of keys assigned to a node are used to digitally sign/verify network requests (see Section 5.2.1 for more details about digitally signed requests) and to calculate the GUID of the node. The GUID of the node is calculated by hashing the public key of the node, so that it is unique to the node that can correctly sign the requests made. This approach is similar to the one used by SFS [197, 203] and other distributed systems, such as OceanStore [112], Tahoe-LAFS [191], and IPFS [113]. This approach adds a security layer to the SOS against *Sybil* attacks. A Sybil attack is one where the attacker attempts to forge a node's identity [219]. In a distributed system, for example, an attacker may forge the identity of multiple nodes with the aim to gain trust in the system and perform malicious operations.

```
{
  "type" : "Node",
  "guid" : <hash(d_public_key)>,
  "d_public_key" : <public key>,
  "hostname" : <hostname> or <IP address>,
  "port" : <port number>,
  "services" : {
    "nms" : {
      "exposed" : <true/false>
    },
    "mdms" : {
      "exposed" : <true/false>
    },
    "urms" : {
      "exposed" : <true/false>
    },
    "mms" : {
      "exposed" : <true/false>
    },
    "sms" : {
      "exposed" : <true/false>
    },
    "cms" : {
      "exposed" : <true/false>
    }
  }
}
```

Manifest 5.2: Node Manifest in JSON format.

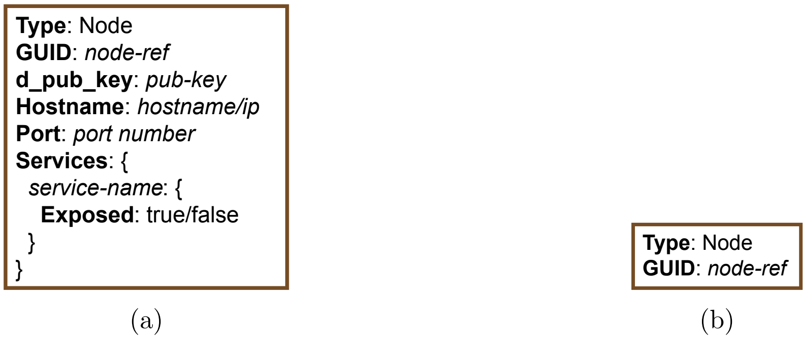
A SOS node may provide any or all of the following services:

- Node management service (NMS)
- Manifests and Data management service (MDMS)
- User-Role management service (URMS)
- Metadata management service (MMS)
- Storage management service (SMS)
- Context management service (CMS)

Each service handles a particular aspect of the model. Section 5.2.1 describes these services in more detail. The implementation of a node should expose each service to the network via a REST API, as discussed in more detail in Section 5.2. The node abstraction, from now on, will be represented using the schematics in Figure 5.1:

5.1.4 The Data Model

The data model defines the data that may be stored in the *Sea of Stuff* and how data is stored and retrieved irrespective of its location; aggregated together; and versioned at different granularities. The data model consists of three types of entities: *atoms*, *compounds*,



```
Type: Node
GUID: node-ref
d_pub_key: pub-key
Hostname: hostname/ip
Port: port number
Services: {
  service-name: {
    Exposed: true/false
  }
}
```

(a)

```
Type: Node
GUID: node-ref
```

(b)

Figure 5.1: Two different graphical representations of the same node manifest, where (b) is a simpler representation of (a).

and *assets*. The next subsections introduce these three entities, what features they provide, their limitations, and how they are represented as manifests within the SOS.

5.1.4.1 Atom

Two of the most basic and important properties of a storage system are how data is represented and how data and data locations are related one another. The *Background* and *Literature Review* Chapters have explored how these two properties are related and some solutions provided by different systems. The *atom* is the data metaphor of the SOS model and also the entity that abstracts data from locations.

An **atom** is an immutable sequence of bytes identified by a GUID, deterministically derived from hashing the atom's sequence of bytes. An atom is associated with an **atom manifest**, which allows the atom to be managed and referenced independently from locations (see Figure 5.2). The atom manifest serves three main functions: (I) finding the atom in the *Sea of Stuff*; (II) checking the integrity of the retrieved data; and (III) aggregating potential locations of the atom. The atom manifest is identified using the same GUID as the atom it describes. However, the atom and the atom manifest are two distinguishable entities, since the former is stored in the SOS data space, while the latter is stored in the SOS manifest space.

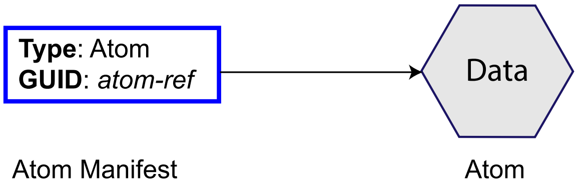


Figure 5.2: Diagram showing the logical relationship between the atom manifest and the atom.

An atom can be stored in one or multiple locations. The SOS data model supports locations that can be accessed through the SOS protocol (sos://) — **SOS locations** — or via other existing access protocols — **external locations** — such as HTTP, HTTPS, FTP, and SFTP.

SOS locations are expressed as URIs and bind together the GUID of the atom and the node where it is stored as follows:

`sos://<host-guid>/<content-guid>`

The **host-guid** is the identifier for a node that can serve the atom matching the *content-guid*. The **content-guid** is the GUID generated by hashing the atom[87], which can be used to verify the integrity of the atom when this is returned by the node identified by *host-guid*.

External locations can be described by multiple schemes, each reflecting its own properties. For instance, HTTP locations are not self-describing, meaning that there exists no relationship between the URI and the content it addresses. For example, the location *http://example.com/image.jpg* may contain an image of a bird, a fish, or not an image at all, and when one attempts to retrieve its content, he/she is unable to verify whether the content has changed or not.

Locations stored in the atom's manifest are marked as: provenance, cache, and persistent, as described below.

- **Provenance**: the original location of the atom.
- **Cache**: a location that contains the atom and has good access properties in terms of latency, throughput, and/or availability.[88]

[87] Remember that this is the same GUID for the atom and the atom manifest.

[88] Note that the term cache, in this instance, is merely an hint to the properties of the location and it is not used in the traditional sense, as described in the *Background* chapter.

- **Persistent**: a location that contains the atom with high durability.

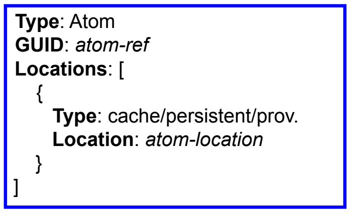
The provenance, cache, and persistent labels serve only as retrieval hints for the atoms stored at the given locations, so that nodes in the SOS can make better decisions when retrieving atoms. The model does not provide any mechanism to enforce the relationship between the retrieval hints and the actual data stored at those locations.

Manifest 5.3 outlines the structure of the atom manifest. The GUID field of the atom manifest identifies both the manifest and the atom. The SOS supports a content-addressable namespace, so the GUID can be used to locate the atom using a DHT (see Section 3.5.1.2). Alternatively, the atom can be found via the atom manifest, which can store a set of locations, making the manifest self-contained and anyone having access to it can retrieve the atom without necessarily having to contact other nodes to discover its location. An atom manifest is represented graphically as in Figure 5.3.

```
{
    "type" : "Atom",
    "guid" : <hash(data)>,
    "locations":
        [
            {
                "type" : "cache", "persistent", or "provenance",
                "location" : <SOS/external location>
            }
        ]
}
```

Manifest 5.3: Atom Manifest in JSON format.

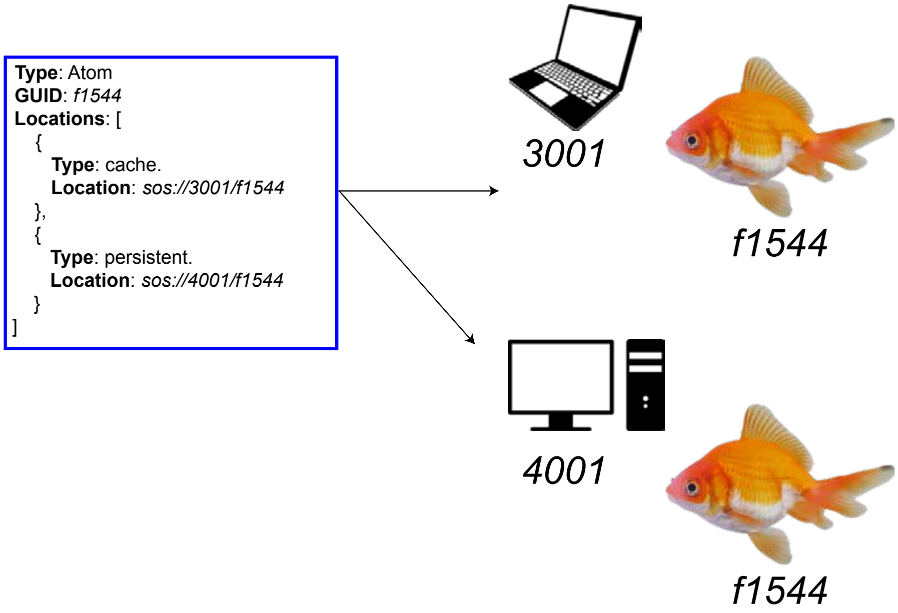
The list of locations stored in an atom's manifest is mutable. However, even though the atom's manifest can change over time, its association with the atom is immutable.



Type: Atom
GUID: *atom-ref*
Locations: [
  {
    Type: cache/persistent/prov.
    Location: *atom-location*
  }
]

Figure 5.3: Atom Manifest representation

The example in Figure 5.4 shows how an atom (*i.e.*, the picture of the fish), can be stored in multiple locations. In this case the locations are in nodes *3001* and *4001* and are abstracted by the atom manifest identified by the GUID *f1544*.



```
Type: Atom
GUID: f1544
Locations: [
  {
    Type: cache.
    Location: sos://3001/f1544
  },
  {
    Type: persistent.
    Location: sos://4001/f1544
  }
]
```

Figure 5.4: Example of an atom manifest containing references to two atoms stored in two different locations (see *Image Credits* for the source of the fish image and the PC and laptop icons).

5.1.4.2 Compound

Another important property of a storage system is its ability to aggregate data together to provide a logical organisation across the system. Hierarchical file systems use directories to collect files and other directories together. Zip files and application packages are also abstractions that serve the purpose of aggregating content. Version control systems, like git [92] and mercurial [93], and P2P storage solutions, such as OceanStore [112], IPFS [113] or Perkeep [190], use manifest-like structures to aggregate content by content-addressable references. Composite formats are also a type of abstraction that permits content to be aggregated with some meaningful semantics. The SOS allows content to be collected together through the compound abstraction using an approach similar to the one adopted by git and IPFS.

A **compound** is an immutable collection of GUIDs that refer to SOS entities, such as atoms and/or other compounds. A compound serves two purposes: (I) aggregating entities and (II) aggregating chunks of a single datum. In the former case the compound is known as a **collection type compound**, while in the latter case it is called a **data type compound**. *Data type compounds* are useful when storing very large amounts of data, which are split into smaller chunks to exploit de-duplication (*i.e.*, two exact chunks are stored only once). Further, moving smaller chunks of data in a distributed system is faster, especially when the chunks are transferred in parallel.

Overall, the compound consists of three main elements: the compound type, its contents, and the GUID that identifies it, derived from the contents. The contents of a compound are organised as a list of references to other SOS entities, such as atoms and compounds. The compound can also be used to aggregate other types of SOS entities, such as versions or metadata, introduced in the next sections. Moreover, each entity contained in the compound can be optionally labelled using an arbitrary string. Labelling the contents of the compound allows content that is content-addressable to be referred by a human-readable name. Finally, the GUID of the compound is calculated by hashing the manifest type, the compound type, and the list of contents that make up the compound itself:

$$GUID = hash(\mathbf{Compound} + \mathbf{T} + \langle compound\ type\rangle + \mathbf{C} + [\langle content\rangle + \mathbf{.}])$$[89]

The elements in **bold** are fixed and necessary to calculate the hashes for all compounds. The fixed elements are necessary to define the structure of the compound entity and avoid second preimage attacks.[90] The elements between the brackets ⟨⟩, instead, change based on the type of compound and its contents. The *⟨content⟩* of the compound is defined as

[89] The list of contents consists of (GUID, label) pairs, which are sorted alphabetically along the GUID values. The sorted pairs are then concatenated together before being hashed.

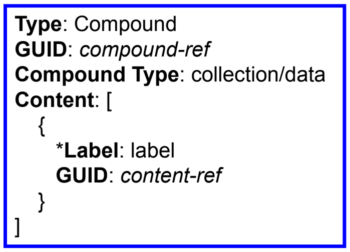
[90] A preimage attack on cryptographic hash functions consists of a malicious party trying to find a sequence of bytes that evaluates to a specific hash value [220]. A Second preimage attack is one where given some input data, it is possible to find another sequence of bytes whose hash value is the same as for the first data input.

either a reference to the content or as a label-reference pair. The purpose of the label is to embed useful information about the associated reference. The ability of the compound to aggregate content by references means that content can be de-duplicated as shown in the example in Figure 5.6. The GUID of the compound can be used to ensure integrity over the compound manifest as well as over its contents.

The elements of the compound are organised in a **compound manifest** (see Manifest 5.4), which can be represented graphically as shown in Figure 5.5.

```
{
   "type" : "Compound",
   "guid" : <hash(type, compound_type, contents)>,
   "compound_type" : "collection" or "data",
   "contents" :
      [
         {
            "label" : <label value>,
            "guid" : <GUID to content>
         },
         {
            "guid" : <GUID to content>
         }
      ]
}
```

Manifest 5.4: Compound Manifest in JSON format.



Type: Compound
GUID: *compound-ref*
Compound Type: collection/data
Content: [
  {
    ***Label**: label
    GUID: *content-ref*
  }
]

Figure 5.5: Compound Manifest representation. The starred label field is optional.

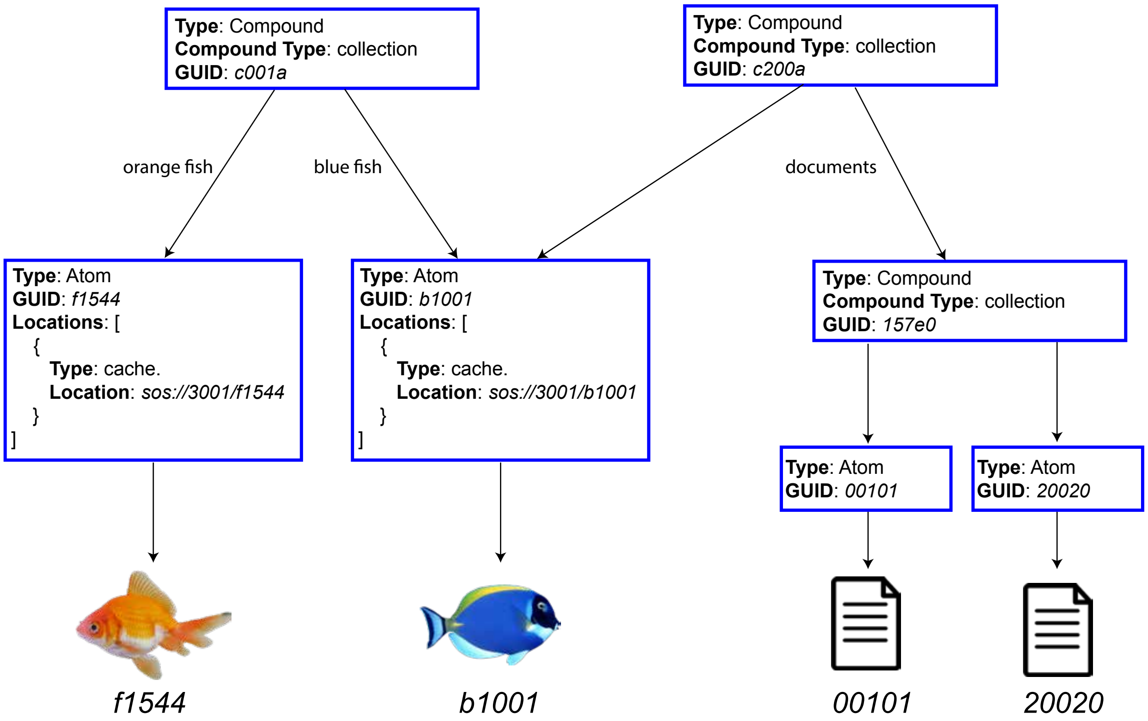


Figure 5.6: Diagram showing an example relationship between compound manifests and their contents. The atom *b1001* (*i.e.*, the blue fish) belongs to the contents of both compounds *c001a* and *c200a* and it is de-duplicated. In addition, it should be noted that some of the compound contents are labelled, while others are not (see *Image Credits* for the source of the fish images and the file icon).

Paths

The result of using compounds to aggregate atoms or other compounds is a directed acyclic graph (DAG) of entities, similar to hierarchies for file systems. This graph can be traversed following the references inside the compounds. The set of references needed to address an entity forms a path (see Section 2.1.2), where each level in the path is delimited by the '/' (slash) symbol. The following is an example of a path to access a specific entity of the compound:

```
<GUID compound>/<GUID entity>
```

Alternatively, the path can be constructed using the label of the entity (if present):

```
<GUID compound>/<label entity>
```

The label must be unique within the compound. Moreover, the same principle can be applied recursively for entities that are compounds, as shown in the following examples, for the compound with *GUID 29abe*:

`29abe/6029a` // Entity *6029a* is a compound labelled *foo*

`29abe/foo`

`29abe/foo/59aeb` // Entity *59aeb* is an atom labelled *bar*

`29abe/6029a/59aeb`

`29abe/6029a/bar`

`29abe/foo/bar`

5.1.4.3 Asset and Versions

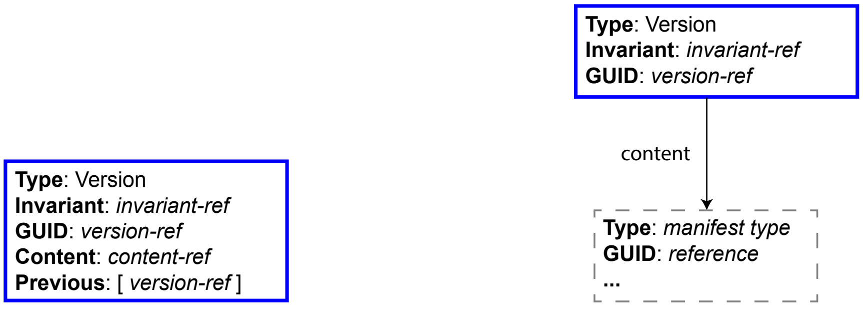
The third entity type of the data model is the **asset**, an abstraction that allows users to manage content irrespectively of where it is stored and how it has changed over time. An asset is a mutable collection of immutable **versions**, which are linked to each other to form a Merkle DAG (see Section 3.3.3.2). Each version refers to some *content*, which can be an atom or a compound. The collection of versions that make the asset is identified by an *invariant*, a GUID that indicates that versions are associated with the same asset. Unlike other GUIDs, the invariant is not derived from the referenced contents, but is randomly generated and valid irrespective of the asset's contents. Versions are linked with each other by having references to their *previous* versions. The GUID of the version manifest is derived by hashing its content and attributes as following:

$$GUID = hash(\textbf{Version} + \mathbf{I} + \langle invariant \rangle + \mathbf{C} + \langle content \rangle + \mathbf{P} + [\langle previous \rangle + \mathbf{.}]$$

A version is represented by a manifest as shown in Manifest 5.5. In the remainder of this thesis, the illustrations in Figure 5.7 will be used to represent the version manifest. In some cases the content reference will be drawn explicitly as in Figure 5.7b, while in other cases the references are omitted as in Figure 5.7a.

```
{
  "type" : "Version",
  "guid" : <hash(invariant, content, previous)>,
  "invariant" : <invariant GUID>,
  "content" : <content GUID>,
  "previous" : [ <previous GUID> ]
}
```

Manifest 5.5: Version manifest in JSON format.



(a)

(b)

Figure 5.7: Two graphical representation of a version manifest.

How Assets Evolve over Time

An asset describes mutable content in the SOS by collecting together all the immutable versions of the content, which are marked with the same invariant GUID. The remainder of this section describes how an asset is created and used to version content.

Asset Creation and Versioning

The creation of an asset is as simple as creating a version manifest with a randomly generated GUID for the invariant and assigning some content to the version by reference. Versions are immutable and do not change, so assets enable mutability over content through the addition of new versions. When this evolution, or versioning, step takes place, a new version must be created such that it has the same invariant of the previous version, a reference to the new content, and a reference to the previous version, so that a relationship between the two versions is established. Figure 5.8 illustrates the creation of an asset and its evolution, with new versions having references to their previous ones.

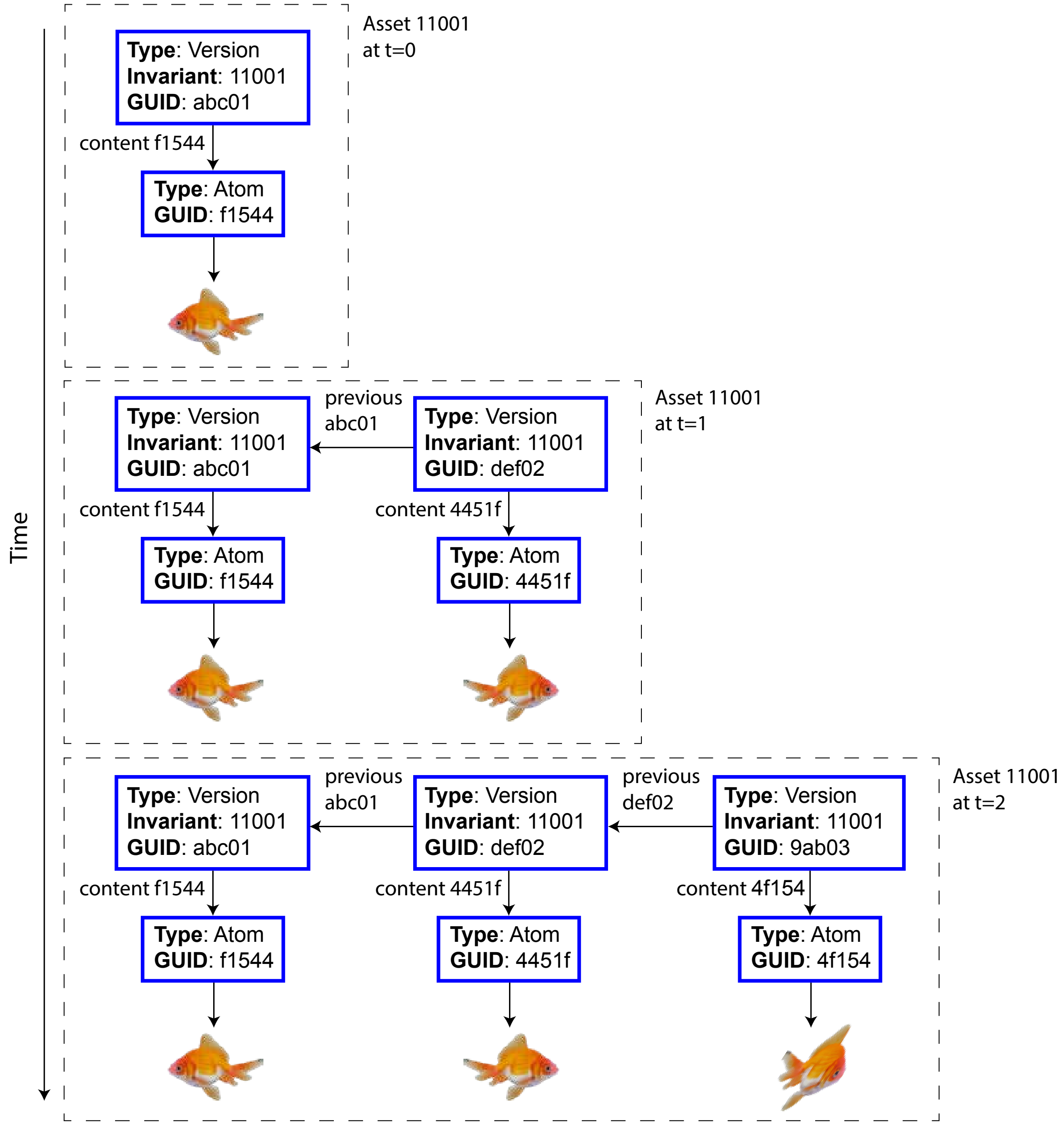


Figure 5.8: Diagram showing the creation of the first version of an asset and how it evolves over time.

Branching

An important operation in VCS is branching, the result of two, or more, versions that evolve from the same version, known as the common ancestor of the branches. Branching represents an important workflow in VCS, since it allows its users to evolve content in multiple ways in parallel. Figure 5.9 illustrates the branching operation for versions *ghi03* and *jkl04* in relation to their common ancestor *def02*. The branching operation increases the number of leaves in the DAG by one.

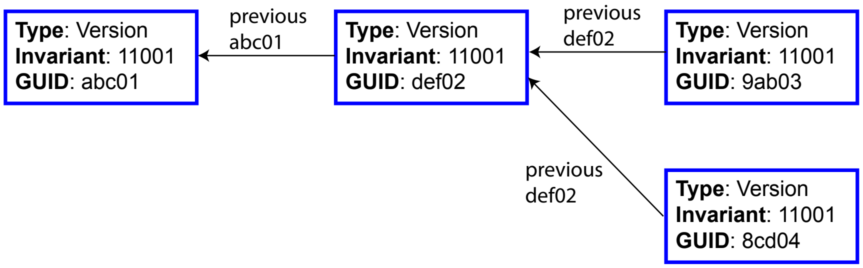


Figure 5.9: Diagram showing a version being branched into versions *ghi03* and *jkl04*.

Merging

Merging is the operation of creating a new version from two or more previous versions. Conflicts resulting from a merging operation are not covered by the SOS model, but tools operating on the SOS can provide conflict resolution mechanisms when needed. Merging can be thought of as the opposite of the branching and can reduce the number of leaves in the DAG if more than one of the previous versions was a leaf. Figure 5.10 illustrates the operation of merging two versions, *ghi03* and *jkl04*, into version *mno05*.

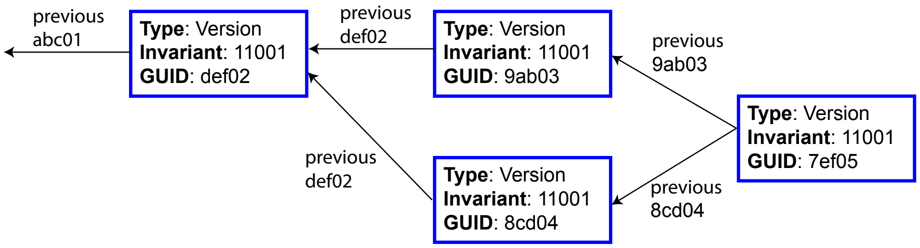


Figure 5.10: Diagram showing two versions being merged into one, *mno05*.

Versions with Compounds of Versions

A property of the model presented in this work is its ability to version content at arbitrary granularities. Data in a VCS is versioned at the repository level (see Section 3.3), such that the history of a file, or folder, is strictly related to the history of the entire repository. In the SOS, instead, it is possible to version data and collections as well as other assets or collections of assets. The example shown in Figure 5.11 illustrates the asset with invariant *959ab* referring to a compound of versions.

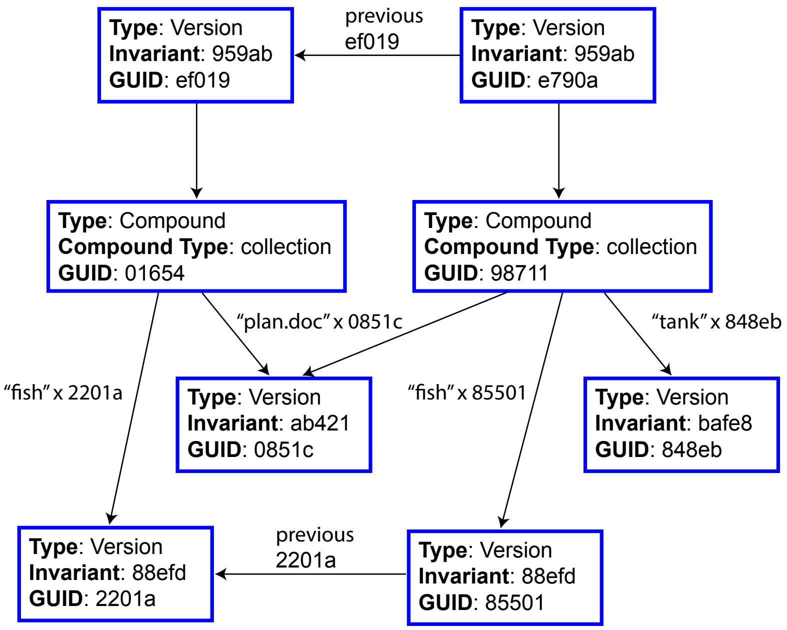


Figure 5.11: Diagram showing how the SOS allows to have versions of compounds which refer to versions of other assets.

Head and Tips

The 'current' version of an asset is defined as its **head**. A client operating on the node can decide to change the head of an asset to any version of the asset. This action affects the state of the client only. The **tips** of an asset are all the versions of an asset that are leaves of the DAG. Tips are very important because versions have references to their previous versions but not to future ones, so it becomes hard to navigate an asset without the tips. A tip can also be defined as the latest version for a given branch of an asset.

The SOS models head and tips, but these are not represented as manifests and are therefore valid only within a node. Thus, the same asset stored in a different node can have different head and tips based on what actions were performed on that node.

Consistency

The versioning mechanism described above supports a loose consistency model, where multiple users or processes can end-up having a different view of the SOS and its data. The SOS data model does not describe, in fact, any consistency mechanism to ensure that two users have the same view of a given asset. Employing a loose consistency approach

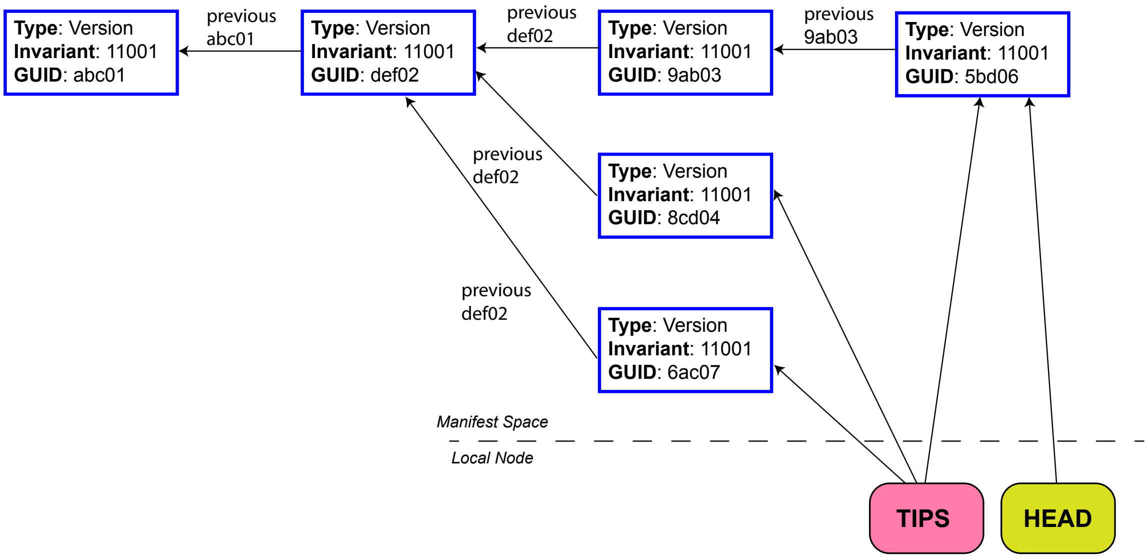


Figure 5.12: Diagram showing the head and tips of an asset's DAG.

results in a simpler data model and allows users and/or applications to build any stronger consistency strategy on top of it. For example, a shared file system that requires strong consistency could be implemented on top of the SOS in such a way that any data updates are serialised by a central service. Alternatively, the nodes providing the shared file system could support a strong consistency model through the implementation of an agreement protocol among them.

5.1.4.4 Summary

The SOS data model allows data to be abstracted from its locations with atoms and atoms' manifests, organised in hierarchies using compounds, and versioned at arbitrary granularities as assets. A very important property of the SOS model is that its entities are immutable and independent and the linking between these entities is achieved through references (*i.e.*, GUIDs). The use of GUIDs gives the model a degree of flexibility over its entities that allows any content to be versioned at different granularities, which is one of the desirable properties of an ideal system where version control is in place, as stated in Chapter 1. Moreover, entities can be used in more ways than the ones suggested in this work. Chapter 9 discusses alternative ways to use the SOS model and how it could be improved in the future.

The diagram in Figure 5.13 summarises the SOS data model when all components are put together. In particular, the diagram shows all the possible relationships among all entities of the data model.

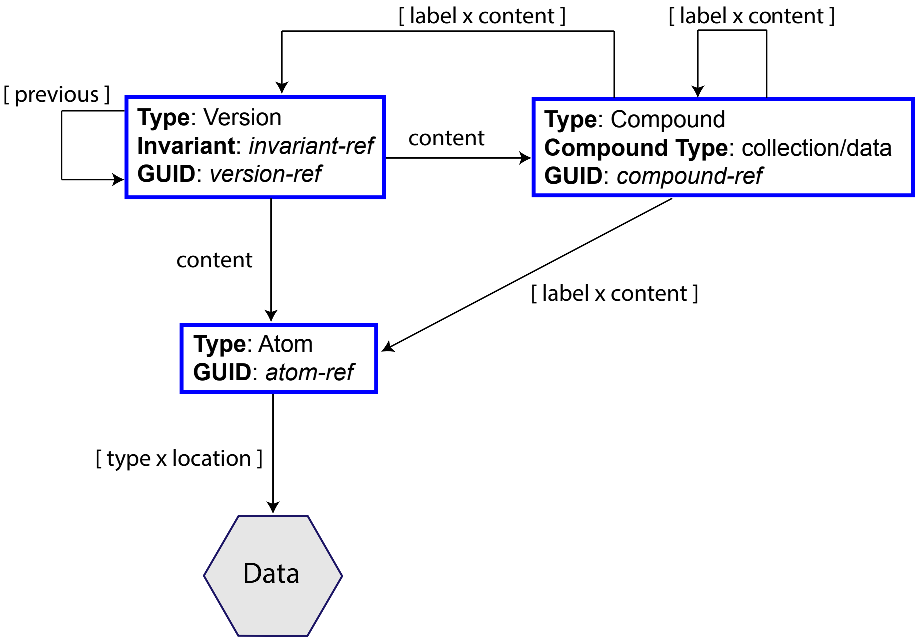


Figure 5.13: Schematic for the data model.

5.1.5 The Metadata Model

The data model enables users to store and access data (*i.e.*, via atoms) irrespective of its locations, aggregate content together (*i.e.*, via compounds), and version content over time (*i.e.*, via assets). The metadata model extends the data model by allowing the content that is referenced by a version to be associated with arbitrary metadata. Section 2.1.3 defined metadata as "data about data" and stated how important metadata is in data storage systems. Chapter 3 explored how state of the art storage systems manage metadata.

In the SOS, metadata is represented through the **metadata manifest** and it is of *extrinsic* type, since it is not stored within the data. The metadata manifest consists of a list of properties (see Manifest 5.6). A **property** is defined as the triplet *(key, type, value)*. The property **key** can be any arbitrary string, the **type** defines the nature of the property, and the **value** is the actual state of the property associated with the given *key* and of the specified *type*. The prototype presented in Chapter 6 supports the types listed in Table 5.1. However additional property types can be added, such as arrays, objects or the *timestamp* type. The type attribute, in fact, allows a property to be extended beyond the standard JSON capabilities.[91] Properties of type *string*, *long*, *double*, and *boolean* are literal values, while *GUID* properties enable metadata to link content over other entities of the SOS, such as the version of another asset or a role (see Section 5.1.6 below).

Type	Examples
String	"text/html", "red"
Long	-1, 0, 1, 255
Double	-3.14, 2.71828
Boolean	true, false
GUID	SHA256_16_5a63f...

Table 5.1: Metadata types currently supported by the current prototype of the SOS.

```
{
  "type" : "Metadata",
  "guid" : <hash(properties)>,
  "properties" : [
```

[91] JSON only supports the following types: string, number, JSON object, array, boolean, and *null*.

```
    {
      "key" : <property name>,
      "type" : "string", "long", "double", "boolean", or "guid",
      "value" : <property value>
    }
  ]
}
```

Manifest 5.6: Metadata manifest in JSON format.

Metadata is linked to the relevant content through the version manifest. The version manifest allows metadata to be linked by using an additional, optional, metadata field (see Manifest 5.7, and Figure 5.14 for an illustration of the relationship). Having the relationship between a version manifest and a metadata manifest stored in the former allows metadata to be de-duplicated across multiple versions. The metadata field, if present, will be used to generate the GUID of the version:

$$GUID = hash(\mathbf{Version} + \mathbf{I} + \langle invariant \rangle + \mathbf{C} + \langle content \rangle + \\ \mathbf{P} + [\langle previous \rangle + \mathbf{.}] + \mathbf{M} + [\langle metadata \rangle + \mathbf{.}])$$

```
{
  "type" : "Version",
  "guid" : <hash(invariant, content, previous, metadata)>,
  "invariant" : <invariant GUID>,
  "content" : <content GUID>,
  "previous" : [ <previous GUID> ],
  "metadata" : <metadata GUID>
}
```

Manifest 5.7: Version manifest linking content and metadata.

The list of metadata properties is generated through a metadata engine that processes the content of the version (more on this in Sections 5.2.2.4 and 6.1). Figure 5.15 shows an example of a *png* fish image versioned by the asset *10295* and described by the metadata manifest identified by the GUID *33745*. The metadata describes the fish atom in terms of

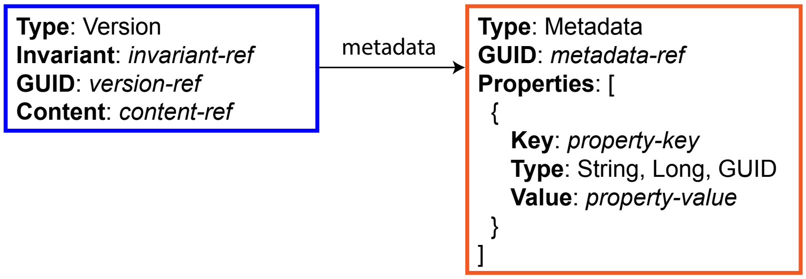


Figure 5.14: Diagram showing the relationship between a version manifest and the associated metadata manifest.

its type (png), its size (855819 bytes), and its most prevalent colour (orange).

A node supporting the SOS model can build an inverted index on the stored metadata manifests and a reverse map for the version-metadata relationship to provide search over the content of the SOS using the information stored in the metadata. Having these two data structures, anyone searching for content of colour 'orange', will find the metadata manifest *33745* from the inverted index and the version *54320* from the reverse map. These data structures are stored locally to the node, but content is still searchable across the SOS as long as nodes expose their functionalities via an API.

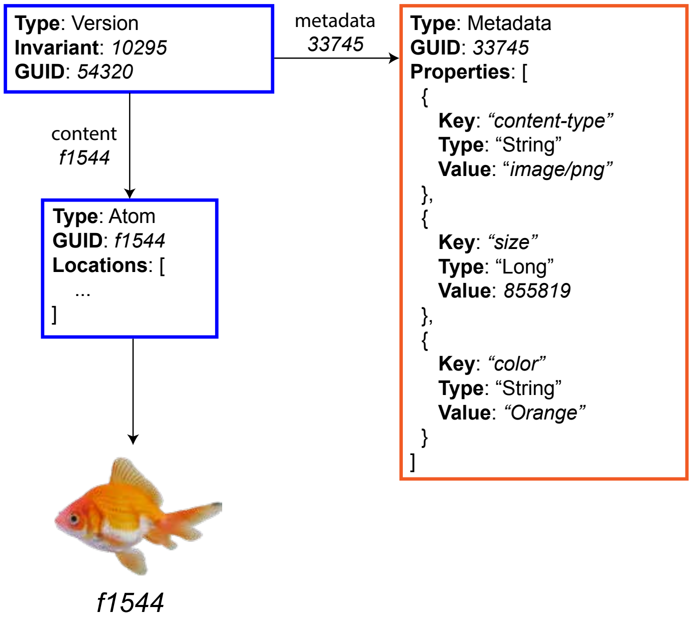


Figure 5.15: Example of metadata being linked to an atom through the version manifest.

Decoupling the metadata from the data it describes through the version manifest has its advantages as well as its disadvantages. For instance, having the metadata stored as a separate manifest from the version manifest, it is possible to describe and handle the content of a version without having to have the actual content. Moreover, it is possible to replicate the metadata independently of the data it describes. On the other hand, getting the data from the metadata has a cost due to the additional index data structures that a node has to maintain.

Alternative Metadata Design

An alternative design for modelling the metadata consists of storing each property in its own manifest and using a compound-like manifest to group together all such properties. The advantages of this design are better de-duplication of the metadata and the ability to perform metadata search without the need of an inverted index. On other hand, this design would add another level of indirection to retrieve metadata for some given version. Such alternative design could be explored in the future.

5.1.6 The User-Role Model

In the SOS, data ownership and protection are achieved through the user-role model. This model consists of two concepts: the user and the role. In the SOS the concept of user is associated with the one of a real-life user that interacts with the SOS. For example, each of the three people ‘Simone’, ‘Al’, and ‘Graham’ can be associated with a respective named user entity. A user interacts with the SOS through one or multiple roles. The role is an SOS entity that users use to: (I) define ownership of content; (II) digitally sign manifests; (III) and protect content via encryption. Examples of user-role relationships are the following:

- The user ‘Simone’ is associated with the ‘Home’ and ‘PhD’ roles;
- The user ‘Al’ with the ‘Home’, ‘Work’, and ‘Photographer’ roles; and
- The user ‘Graham’ with the ‘Home’, ‘Work’, and ‘Runner’ roles.

The User

The relevant attributes of the **user** entity are its name and a pair of asymmetric keys, used to digitally sign/verify content. The name is a human-readable string that allows SOS users to informally refer to end-users. The private key is used for the creation of all the roles associated with the user (see next Section), while the public key is used to verify such roles. The generic scheme of the **user manifest** is shown in Manifest 5.8.

```
{
  "type" : "User",
  "guid" : <hash(type, name, d_public_key)>,
  "name" : <User name>,
  "d_public_key" : <public_key> # public key for digital signatures
}
```

Manifest 5.8: User Manifest.

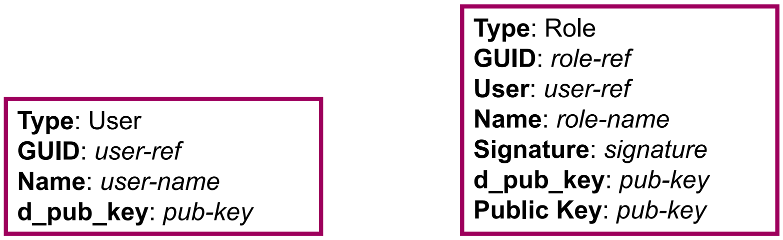
The Role

The role entity consists of two key pairs, one to digital sign manifests, as explained in Section 5.1.6.2, and one to protect content, as explained in Section 5.1.6.3 (see Manifest 5.9). A role is always associated to a user and signed using the user's private key.

```
{
  "type" : "Role",
  "guid" : <hash(type, user, name, signature, d_public_key, public_key)>,
  "name" : <Role name>,
  "user" : <User GUID>,
  "signature" : <as signed by User>,
  "d_public_key" : <public_key>, # public key for digital signatures
  "public_key" : <public key> # public key for content protection
}
```

Manifest 5.9: Role Manifest.

The user and role manifests will be represented as in Figure 5.16. The three functionalities provided by the role — ownership, manifest signature, and content protection — are discussed in more details in the next subsections.



Type: User
GUID: *user-ref*
Name: *user-name*
d_pub_key: *pub-key*

(a)

Type: Role
GUID: *role-ref*
User: *user-ref*
Name: *role-name*
Signature: *signature*
d_pub_key: *pub-key*
Public Key: *pub-key*

(b)

Figure 5.16: Diagrams for the user and role entities.

5.1.6.1 Ownership

Content ownership is established by referencing a role from the metadata of a given version. Moreover the metadata can be used to store not only the owner of a version, but also other types of role relationships (*e.g.*, author, tester, client, stakeholder, *etc.*).

```
{
  "type" : "Metadata",
  "guid" : <hash(properties)>,
  "properties" : [
    {
      "key" : "owner", # This could also be author, tester, client, stakeholder, etc.
      "type" : "guid",
      "value" : <reference to role>
    }
  ],
  ...
}
```

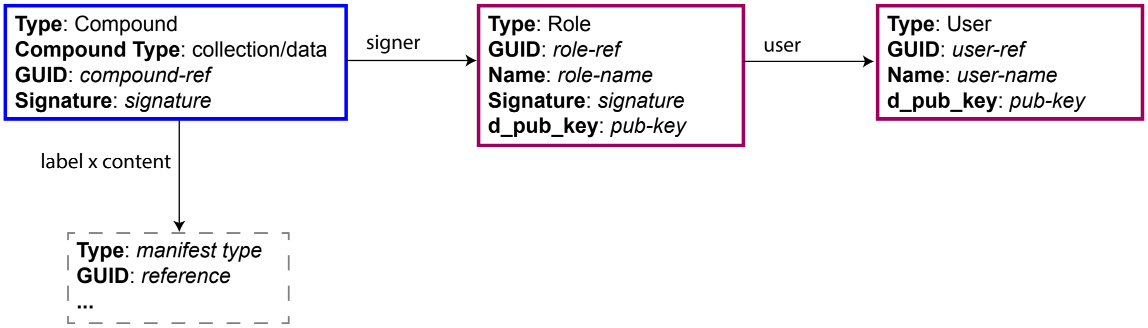
Manifest 5.10: Metadata manifest in JSON format containing the *owner* property.

5.1.6.2 Signing Manifests

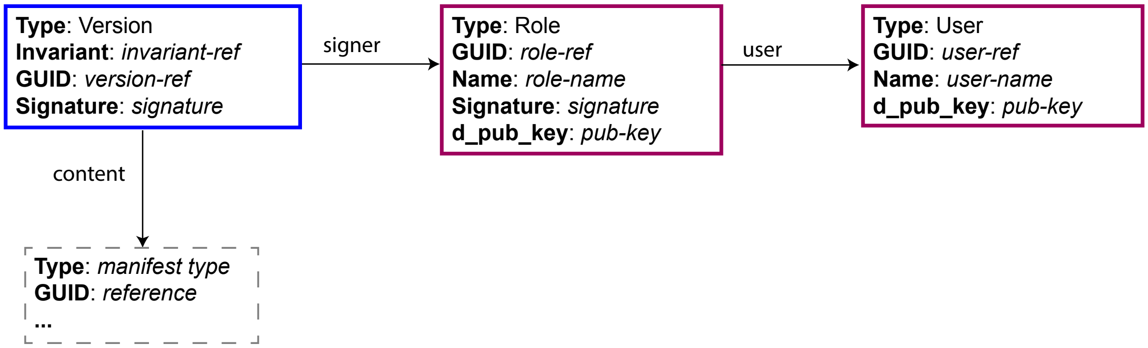
The role entity allows manifests to be digitally signed and therefore to have authentication and non-repudiation, in addition to integrity, over the content of the manifest. The atom manifest acts only as a means to abstract data from its locations, so ensuring data integrity via the GUID field is enough. In the case of the other manifests, however, the digital signature can be used to demonstrate the authenticity of its content. For example, if the role ‘PhD’ by the user ‘Simone’ digitally signs a certain compound manifest, then anyone can verify that the manifest was signed by the role ‘PhD’ and that no one has tampered with it. It should also be remembered that the ‘PhD’ role manifest itself is signed by the user ‘Simone’, so the same properties can be verified for the role manifest. Figure 5.17a shows the relationship between a compound manifest, its role signer and its matching user. A signed manifest is created by adding the *signer* and *signature* fields to it (see Manifest 5.11).

```
{
  "type" : <Manifest Type>,
  "guid" : <hash(Content to Hash)>,
  "signer" : <guid of signer role>,
  "signature" : <sign(Content to Sign)>, # The content to sign depends on the type of manifest
  ...
}
```

Manifest 5.11: Base Signed Manifest JSON structure.



(a)



(b)

Figure 5.17: Diagrams for signed manifests. (a) illustrates a compound being signed, while (b) shows an analogous relationship for a signed version manifest.

5.1.6.3 Protecting Content

Manifests and atoms, as seen so far, are stored in clear text. Hence, anyone who has access to them is able to read them. The role entity allows manifests and atoms to be protected using a combination of symmetric and asymmetric encryption.

Protecting Content

Content in the SOS is protected by encrypting it using a symmetric encryption mechanism (*e.g.*, AES). The symmetric key is randomly generated and then encrypted using the public key of the role. The pair composed by the role GUID and the encrypted symmetric key is then added to the manifest of the protected entity as shown in Manifest 5.12. The generic algorithm to protect content in the SOS using roles is described in pseudo-code in Algorithm 1. In the SOS, protection can be applied at different granularities because not all entries have to be necessarily encrypted. For example, one may want to protect atoms and compounds, but not the metadata describing them. The content to encrypt depends on the type of entity:

- *Atom*: the actual atom is encrypted (see example shown in Figure 5.18 and the resulting protected manifest at Manifest 5.13).
- *Compound*: the list of contents in the manifest is encrypted.
- *Metadata*: the list of properties in the manifest is encrypted.

```
{
  "type" : Protected Manifest Type,
  "guid" : <hash(contents)>,
  ...,
  "keys" : [ {
    "key" : <encrypted key>,
    "role" : <role guid>
  } ]
}
```

Manifest 5.12: Protected manifest in JSON format.

```
{
  "type" : "Atom Protected",
  "guid" : "54842", # hash(encrypted atom)
  "locations" : [ {
       "type" : "persistent",
       "location" : "sos: //09102/54842"
  } ],
  "keys" : [ {
       "key" : "192a+=" # <encrypted(symmetric_key)>,
       "role" : "09af9"
  } ]
}
```

Manifest 5.13: Example of a protected atom manifest in JSON format.

```
Algorithm 1: Generic algorithm to encrypt any content in the SOS through the role.
  Input: Content $C$ to protect
  Result: Encrypted content $C'$ and encrypted key $K_R$
  begin
      1. Generate a random symmetric key $K$
      2. Encrypt $C$ with $K \rightarrow C'$
      3. Encrypt $K$ with the public key of role R, $\mathrm{R}_{\mathrm{pub\_k}} \rightarrow K_R$
      4. Store the manifest with the pair: $(K_R, \mathrm{R})$.
         The manifest can store multiple (key, role) pairs.
      5. if C is an atom, then store $C'$
      6. if C is not an atom, then embed $C'$ as part of the manifest
  end
```

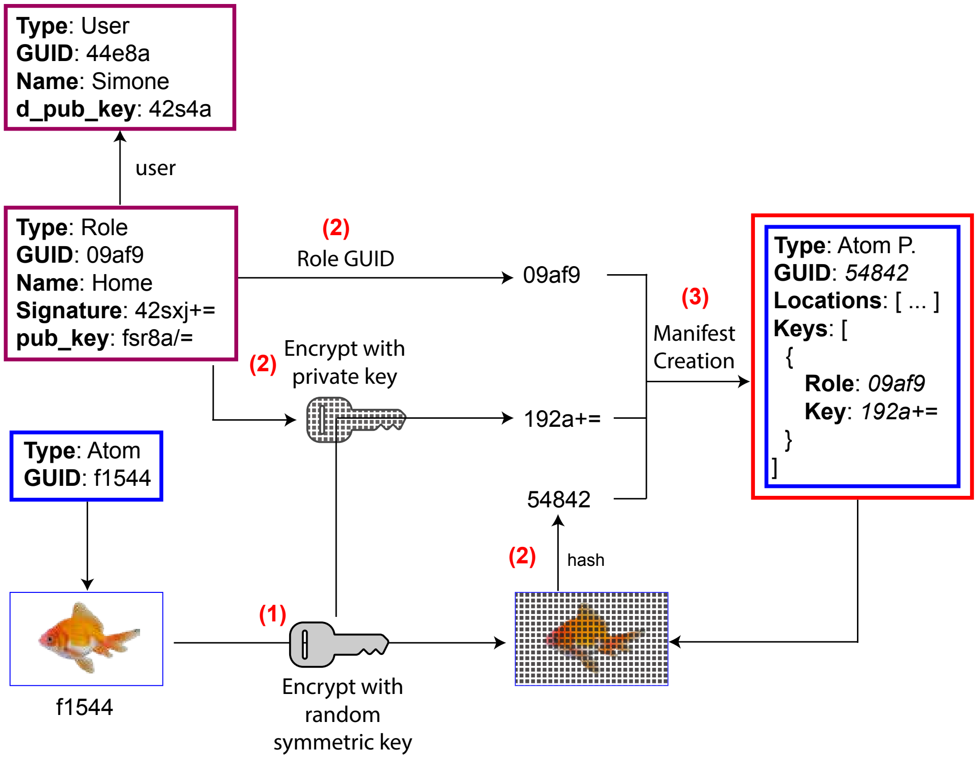


Figure 5.18: Example of an atom being protected through the role entity.

Reading Protected Content

Protected content can be read only by the roles that are listed in the *keys* field of the protected entity manifest. If the role is listed, then it must decrypt the associated encrypted key using its private key and use the decrypted key, which is the same symmetric key used to protected the content, to read the protect entity. The generic algorithm to read protected content using roles is described in pseudo-code in Algorithm 2.

Granting Access

The process of a role (granter) **granting access** to a protected entity to another role (grantee) involves three steps:

- The encrypted key associated to the granter should be decrypted using the granter private key.
- The decrypted key should be re-encrypted using the public key of the grantee.
- The grantee's GUID and the new encrypted key should be added to the manifest (in the *keys* field).

Algorithm 2: Generic algorithm to read encrypted content through a role.

```
Input: Encrypted content $C'$ to read
Result: Decrypted content $C$
begin
    1. Get the manifest $M$ for $C'$
    2. Get the key $K_{R_i}$ for the role $\mathrm{R_i}$ from $M$, where $i$ is one of the roles stored in the key
       set of the manifest
    3. Decrypt $K_{R_i}$ with the private key of R, $\mathrm{R_{priv\_k}} \rightarrow K$
    4. Decrypt $C'$ using $K \rightarrow C$
end
```

5.1.7 The Context Model

One of the challenges identified in the *Introduction* Chapter is how to enable users to control data in a distributed system in an automatic manner. The *Sea of Stuff* provides a computational model, known as **context model**, that allows users to define rules to automatically capture content of the SOS and perform actions over a set of nodes. The context model is based around the concept of **context**, defined as:

> *a set of information used to characterise and automatically manage a collection of related data in a distributed system.*

Contexts are used to organise, search, and share data and collections of data. The following are some examples of how contexts can be used to automatically manage content in the *Sea of Stuff*:

- To classify all images taken with a particular camera, with a high percentage of blue colour and taken during the summer.
- To classify content, as in the previous example, and ensure that such content is replicated at least N-times over some nodes that one can trust.
- To migrate content from one node to another.
- To distribute data partitioned in shards over a set of nodes with a given dispersion factor.

The context model not only allows computation to be performed over the SOS, but also enables end-users to understand and manage their content in new ways. Contexts provide the following three main functionalities:

1. Assets' versions, distributed over a set of nodes, can be automatically classified.
2. A set of actions can be defined and run automatically for the classified versions, as in 1. The operations can be extended (or restricted) to have effect over a set of nodes.

3. The state described by 1 and 2 is automatically enforced over time with eventual consistency.

The implication of using contexts to organise and manage data in a distributed system is that end-users do not perceive their collections of data as static (*e.g.*, compounds), but rather as collections that can mutate over time and act upon changes of the system. Contexts can also be thought of as smart folders[92] over a distributed system with the ability to automatically apply operations over the data.

5.1.7.1 The Context

A context is defined by its predicate and policies, **computational work units** that operate over the content of the SOS. A context has one predicate and zero, one or more policies. The **predicate** is a boolean-valued function that characterises data stored within a **domain** of nodes. All the contents identified by the predicate of the context within its domain are known as the **contents of the context**. A **policy** consists of a set of operations performed to automatically manage the content identified by the predicate over a codomain of nodes. The **codomain** is the collection of nodes over which policies operate.

A context is represented by a manifest containing (see Manifest 5.14):

- A GUID that identifies it.
- A reference to the predicate used to identify its content.
- The list of node references that make the domain of the context.
- A reference to the policies used to automatically manage the content identified by the predicate.
- The list of node references that make the codomain of the context.
- A reference to a compound of contents identified by the predicate (*i.e.*, a new version of the context is created when its contents change).

[92] The concept of smart folders is available in Mac OS and Windows operating systems. Linux users can use FUSE based file systems or other solutions to achieve similar results.

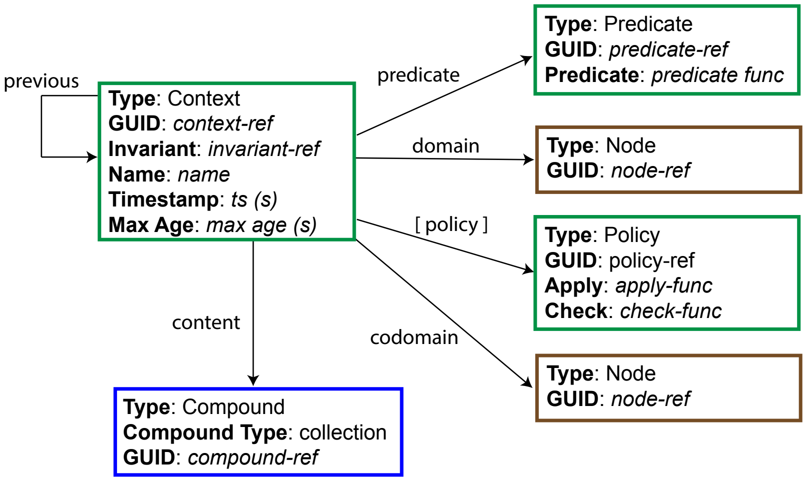


Figure 5.19: Relationship between the context and its components.

Figure 5.19 illustrates the relationship between the context and its components. The GUID of the context is calculated as following:

$$\begin{aligned} GUID = hash(\mathbf{Context} + \mathbf{N} + \langle name \rangle + \\ \mathbf{I} + \langle invariant \rangle + \mathbf{P} + \langle previous \rangle + \\ \mathbf{C} + \langle content \rangle + \\ \mathbf{DO} + \langle domain \rangle + \mathbf{CO} + \langle codomain \rangle \end{aligned}$$

The fields used to calculate the GUID of the context are also indicative of the components that can change over time: the name, the contents of the context, the domain, and the codomain.

The contents of a context can change over time based on the nature of the predicate and/or when assets are added, updated, and deleted in the domain. In order to maintain the immutability property of a context, the context manifest is extended with a *previous* attribute, like that provided by a version manifest. Sections 5.1.7.2 explains in more details how the context is versioned.

```
{
   "type":  "Context",
   "guid":  <hash(type, name, invariant, previous, content,
              domain, codomain, predicate, max_age, policies)>,
    "timestamp":  <timestamp>, # in seconds
    "name":  <context name>,
    "invariant":  <hash(type, predicate, policies, max_age)>,
    "previous":  <previous GUID>, # Optional
    "content":  <content GUID>,
    "domain":  {
        "type":  "LOCAL", "SPECIFIED", or "ANY",
        "nodes":  [ <node GUID> ]
    },
    "codomain":  {
        "type":  "LOCAL", "SPECIFIED", or "ANY",
        "nodes":  [ <node GUID> ]
    },
    "predicate":  <predicate GUID>,
    "max_age":  <maximum age>, # in seconds
    "policies":  [ <policy GUID> ]
}
```

Manifest 5.14: Context Manifest in JSON.

5.1.7.2 The Predicate

The purpose of the context's predicate is to classify SOS content and is expressed as a boolean-valued function that operates over the GUID of an asset's head and returns either true or false. The nodes over which the predicate of a given context is run is called the **domain**. For simplicity, we say that the predicate is run over the contents of the domain to mean that it runs over all the heads of the assets stored within the domain.

The predicate is represented as a manifest (see Manifest 5.15) and it is defined by three components: the *type*, the *GUID*, and the actual predicate function. In the current design the predicate function is defined as a Java method that takes the GUID of the asset's head as a parameter and returns a boolean.

```
{
  "type" : "Predicate",
  "guid" : <hash(predicate)>,
  "predicate" : <code>
}
```

Manifest 5.15: Predicate Manifest.

The Predicate Function

An example of predicate function is the *acceptAll* predicate, which may be used to classify all the assets within a domain. Code 5.16 shows the implementation of the *acceptAll* predicate in Java.

```java
boolean acceptAll() {
    return true;
}
```

Code 5.16: Predicate *acceptAll* expressed in Java.

More commonly, however, a predicate takes the GUID of an asset's head and additional parameters as input. Given the GUID of a version (*i.e.*, the asset's head), the predicate can retrieve and process any SOS entity that is related to it, such as its content, its metadata, and even its previous versions. An example of a predicate that inspects the version's metadata is the *contentType* predicate (see Code 5.17) which retrieves the metadata associated to the version, searches for any metadata property with key "Content-Type" and checks whether its value is the one specified. Txt, jpeg, mp3, doc, or html are all examples of content types that this predicate could operate on.

```java
boolean contentType(GUID guid, List<String> matchingContentTypes) {
    Property property = getProperty(guid, "Content-Type");
    if (property != null) {
        String contentType = property.getValue();
        return matchingContentTypes.contains(contentType);
    } else {
        return false;
    }
}
```

Code 5.17: Predicate *contentType* expressed in Java.

The following are examples of predicates:

- A function that returns true if the version references an atom that is of JPEG type.
- A function that returns true if the version references an atom that is of JPEG type and it is mostly blue.
- A function that returns true if the version references an atom that is of JPEG type and has trees in it (recognised via AI or machine learning algorithms).
- A function that returns true if the version references an atom that is associated with metadata with a certain 'ownership' field.
- A function that returns true if the version is signed by a certain role.
- A function that returns true if the data is stored at a given node (within the domain).

The implementation details about predicate functions are discussed in more details in Chapter 6.

For the purpose of this first design of the SOS, a boolean-valued function was chosen because it discretely divides the managed content in two spaces, one where content is associated with the predicate's context and one where this is not the case. Moreover, thinking about boolean-valued functions and acting upon them is inherently easier.

Future work should explore and evaluate the possibility of using predicate functions that return integer and/or natural numeral values, probability values, sets of values, or even object values. The challenge with using more complex predicates is understanding what the output values mean, how the SOS space is divided, and what policies to apply for the different partitions.

The Time-Validity of a Predicate

The contents of a context can change over time if (I) the contents within the domain change (*e.g.*, a new asset is added or an existing one is updated) and/or (II) the nature of the predicate depends on time. To avoid reprocessing contents that have already been assigned to the context, a *maximum age* attribute can be set, which defines for how long the results of a predicate should be considered valid. The maximum age attribute can take three values: zero, infinite, and a positive integer.

- A context with **maximum age of zero** indicates that its contents are valid only at the time when they were associated to the context, as specified by the timestamp of the context.

- A context with **maximum age of infinity**, or a very large number, will have its contents valid forever (or for such a very long time that the validity of its contents can be ignored). It follows that the results of a predicate can be re-used and there is no need to re-run the predicate over content that was already classified.

- When the **maximum age is set to a value of *N***, then the contents of a context are considered valid only for *N* seconds from the timestamp.

Maximum Age is N seconds – An Example

In the example in Figure 5.20, a context named ‘All JPEG Images’ classifies all the JPEG content stored at the local node. The context has a maximum age of 45 seconds, while its predicate is run every 30 seconds. The context at *t=0* has its contents collected in the compound with *GUID 73969.* These contents are valid for any future context that has a timestamp within *t=0* and *t=45*. Thus, the contents for the context at *t=30* is the pair of the compounds with *GUID a5f4e* and *GUID 73969*. However, the same is not true for the context with *t=60*. The same logic is applied to the context with *t=90*, whose contents are the ones from compounds with *GUID 8ea11* and *GUID 8771c*.

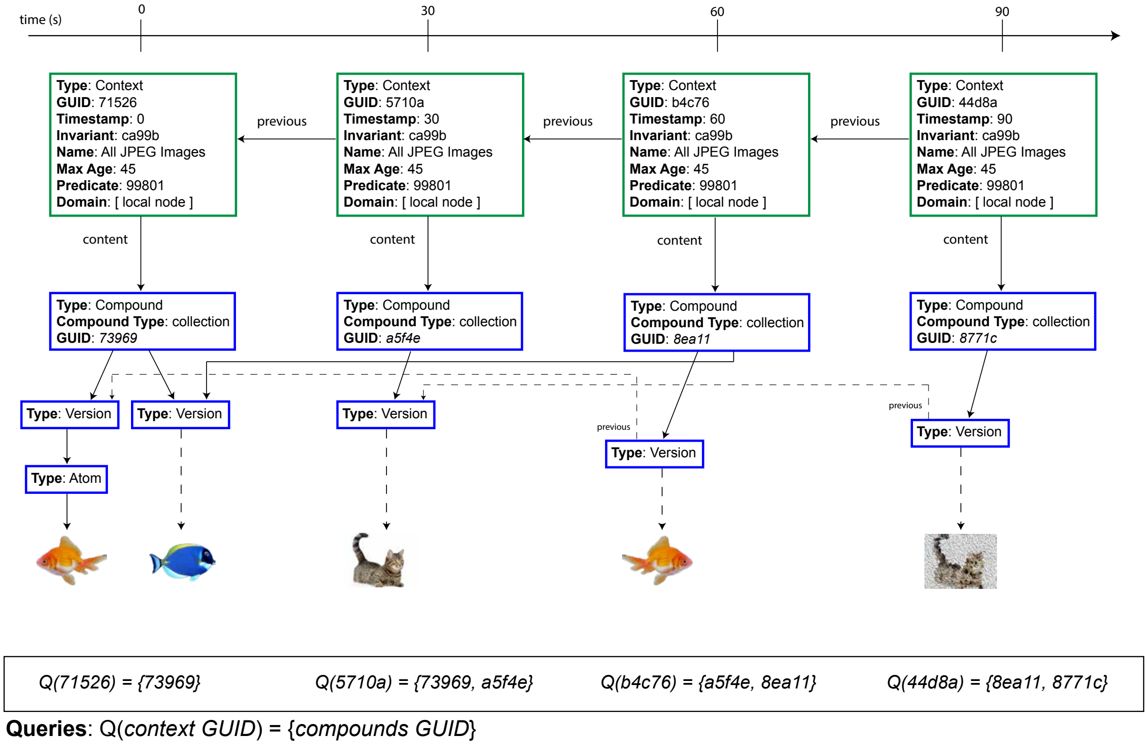


Figure 5.20: The context named ‘All JPEG Images’ has a maximum age of 45 seconds. Some of the intermediary manifests and details are omitted for better readability. The queries at the bottom indicate the compounds returned when retrieving the contents of the different context versions.

Context Synchronisation

The predicate is run over all the contents stored in each node of the domain. When the domain of a context is composed of more than one node, the context and its computational units are copied over the domain, so the predicate is actually run by each node of the domain over their respective contents. In the case of nodes in the domain that do not support contexts (*e.g.*, nodes acting as storage only), the data to be processed is moved across the other nodes in the domain. The higher cost of moving data to nodes that can run predicates should be taken into account when running predicates over multiple nodes.

Each copy of the context is called a **context instance**. The synchronisation between the different instances is performed periodically and relies on the fact that their states will eventually be consistent.

5.1.7.3 The Policy

A policy consists of a set of actions run over the content identified by the predicate. The scope of action of a policy is defined by the codomain of its context. A policy consists of an apply procedure and a check function (see Manifest 5.18). The goal of the **apply** procedure is to change the state of the SOS, for example by replicating content. The **check** function, on the other hand, returns *true* if the state enforced by the apply procedure is satisfied or *false* otherwise.

```
{
  "type" : "Policy",
  "guid" : <hash(apply, check)>,
  "apply" : <code>,
  "check" : <code>
}
```

Manifest 5.18: Policy Manifest.

The Apply Procedure

The purpose of the apply procedure is to perform some changes over the codomain of the context for a given version. Examples of apply procedures are:

- Replicate atoms with a replication factor *N*.
- Compress the atoms that are referenced by the version processed.
- Replicate only the manifests of a version and its components (no atom transfers) with a replication factor *N*.
- Migrate atoms from one node of the codomain to another node, always within the codomain.

- Create a new version of the atom, with a different format (*e.g.*, from jpeg to png).
- Generate multiple atoms with different data formats (*e.g.*, to generate videos of different quality for different screen sizes and/or device).

Code 5.19 is an example of the apply procedure to replicate data by some given factor over the codomain expressed in Java.

```java
void replicate(NodesCollection codomain, GUID guid, int factor) throws PolicyException {
    try {
        Manifest version = getManifest(guid);
        Manifest content = version.content();
        if (content.getType() == ATOM) {
            NodesCollection nodes = filter(codomain, NodeType.Storage);
            addAtom(content.getAtom(), nodes, factor);
        }
    } catch (Exception e) {
        throw new PolicyException("Unable to replicate atom properly");
    }
}
```

Code 5.19: Policy apply function to replicate atoms over the codomain. This function is expressed in Java code.

The Check Function

The check function is a boolean-valued function that verifies that the state of the SOS satisfies the conditions enforced by the apply function within the specified codomain for a given asset's version. Examples of check functions are:

- Atoms are replicated at least N-times over the codomain.
- Atoms are compressed.
- Manifests only (no atoms) are replicated at least N-times.
- Atoms are migrated from a given node to another one.

Code 5.20 is an example of the check function that checks that the number of replicas over the codomain is correct.

```java
boolean isReplicated(NodesCollection codomain, GUID guid, int factor) throws PolicyException {
    try {
        Manifest version = getManifest(guid);
        Manifest content = version.content();
        if (content.getType() == ATOM) {
            int numberOfReplicas = 0;
            for(Node node:codomain) {
                if (nodeHasData(node, guid)) {
                    numberOfReplicas++;
                }
            }
            return numberOfReplicas >= factor;
        }
    } catch (Exception e) {
        throw new PolicyException("Unable to check if atom is replicated properly");
    }
}
```

Code 5.20: Policy check function that checks whether data is correctly replicated over the codomain. This function is expressed in Java code.

Number of Policies

A context can have zero, one, or more policies. The number of policies determines the behaviour of the context, how it affects the rest of the *Sea of Stuff*, and the time to run the context:

- **Zero policies**: the context is used simply to classify and aggregate data via the predicate.
- **One policy**: the context applies the policy to the data associated with the context over the codomain. The cost of running the context depends on the cost of running its predicate as well as its policy.
- **Multiple policies**: the context defines a richer and more complex behaviour by applying multiple policies. Policies are executed sequentially in the order specified

by the context definition. When a context has more than one policy, there is a possibility for conflicts to occur (*e.g.*, one policy replicates content while another one deletes it).

Conflicting Policies

One of the main challenges when designing policies is the ability to avoid conflicts between them. Conflicts results from contradicting policies, belonging either to the same context or different ones, that operate over the same content (in the case of two different contexts, their codomains must overlap). For example, take into consideration the following three policy apply procedures:

- *P1*: Replicate content at least three times over nodes A, B, C, and D.
- *P2*: Replicate content at most twice over nodes A, B, and E.
- *P3*: Replicate content at most twice over nodes F, G, and H.

In this case, an entity that is replicated at least three times by *P1*, then it is forced to be replicated at most twice by *P2*, thus invalidating *P1*. *P3* does not conflict with *P1* or *P2* because it operates on a disjoint codomain (nodes F, G, and H). In the current design of the SOS, conflicts are avoided by limiting the scope of what policies can do.

Nonetheless, restricting the scope of action of policies also limits the overall usefulness of the contexts as a way to automatically manage data in a distributed system. For example, the delete operation is one that can lead to conflicting policies, but it can also allow a context to control how data moves within the codomain.

5.1.7.4 Failure Handling

High availability and reliability are key properties of a fault tolerant distributed system that can be achieved through redundancy. In the SOS, contexts can be used to make contents resilient against faults. Examples are contexts with policies that replicate data a certain number of times or policies that apply error-correction techniques (*e.g.*, Hamming code) over the stored data. Not only policies can be used to build a better fault tolerant system, but the policies themselves are designed to detect failures via the check functions. For instance, a check function for a data replication policy is able to verify that data is replicated according to the rules specified by the policy (see Section 5.2.2.3).

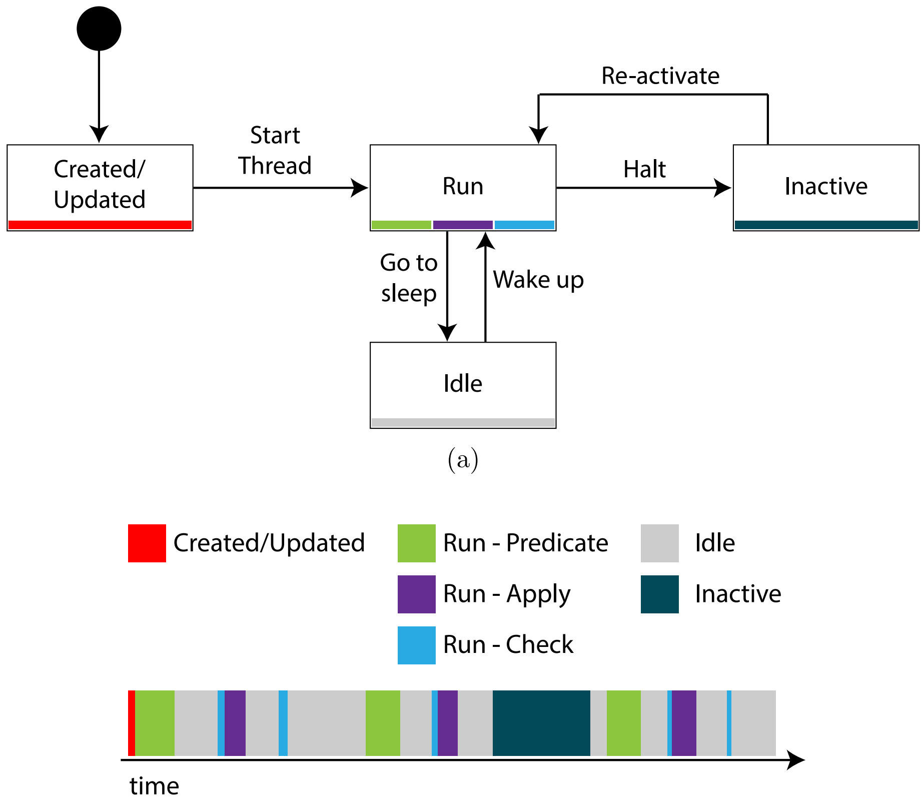
Finally, the computational work units of a context are run on each node of the defined domain independently of the state of any other node in the domain. Thus, the failure of a node does not affect the rest of the nodes in the same domain in terms of what data is identified by the context, via the predicate, and what changes are made over the codomain, via the policies. However, the synchronisation of the instances of the context can be affected and consistency is eventually reached when all the nodes in the domain run correctly.

5.1.7.5 Context Life Cycle

The life cycle of a context and its computational work units is described by the following four states (see Figure 5.21a):

- **Created/Updated**. This state is defined by a context that has just been created or updated (*i.e.*, a new version of the context is created).
- **Run**. A context is in this state when at least one of its computational work units is being executed.
- **Idle**. This is the state between the runs of the computational work units.
- **Inactive**. A context becomes inactive by explicit action of a user. An inactive context is not run until it is re-activated by the user.

Figure 5.21b projects the context life cycle reported above over time. Different colours are used to highlight the different states of the context. After the context is created or updated (in **red**), the run state is reached. A context is in its run state whenever its predicate (in **green**), policy-apply (in **violet**), and policy-check (in **azure**) functions are run. The context is idle (in **grey**) in between all the runs of its computational work units. No computation is run while the context is inactive (in **dark green**).



(a)

(b)

Figure 5.21: (a) Diagram showing the life cycle of a context; (b) timeline showing the life cycle of a context.

In the current implementation of the SOS, presented in the next chapter, the predicate, apply, and check functions are scheduled on a periodic basis and independently of one another so that the execution of one phase is not dependent on the execution of another phase. Moreover, computational units can be run in parallel, improving the throughput of the contents processed. Finally, the life cycle has been designed so that the check function of a given context is always run just before its apply procedure is executed, so that it is possible to establish whether the policy is satisfied or the apply procedure needs to be run.

5.1.8 Summary of the Sea of Stuff Model

This section has presented the *Sea of Stuff* model in terms of its five sub-models:

- **Node model**: defines nodes, the access points to the SOS.
- **Data model**: abstracts data from locations, how it is collected together, and how it is versioned.
- **Metadata model**: describes how data or collections are enriched with metadata.
- **User-Role model**: defines the user-role relationship and how the role entity is used to sign and protect SOS entities.
- **Context model**: describes computation that can be run over multiple nodes in order to manage data automatically.

Figure 5.22 illustrates all the different types of entities of the SOS model. Each sub-model has a very precise set of functions within the overall model, as explained so far. For example, data can only be addressed via atoms and manifests signed by roles only. However, new relationships between different components of the model can also be defined since all entities are identified by GUIDs of the same namespace (see *Conclusions and Future Work* Chapter).

The ability of 'mixing' the components of the SOS model to design new systems and/or features is at the core of the SOS itself. The model presented in this work should not be taken as the definite answer to all the storage problems of today and the future. Instead, the author of this thesis suggests the SOS as a starting point for solving such problems, thus the ability to modify and evolve the proposed model is fundamental.

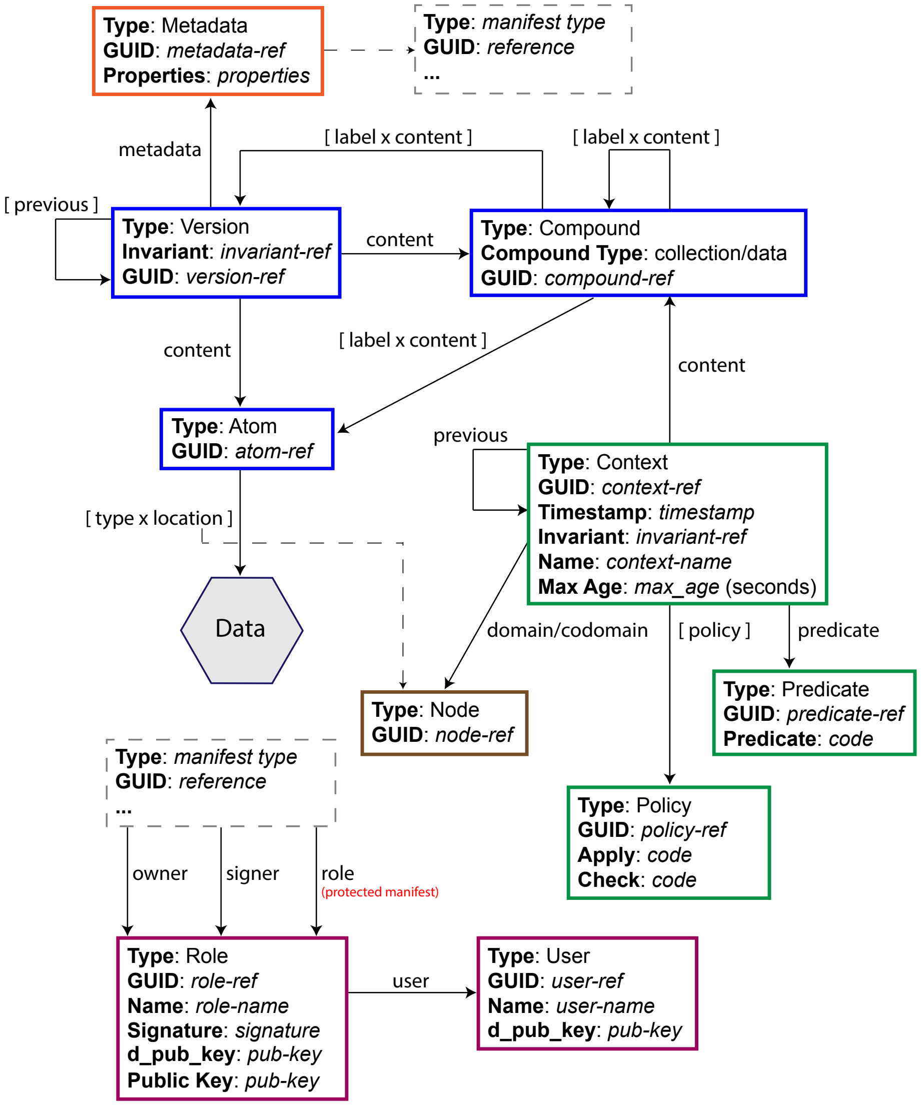


Figure 5.22: Overview diagram of the SOS model.

5.2 The Sea of Stuff Architecture

This section explores the architecture that characterises the structure and behaviour of the *Sea of Stuff* as a distributed system of nodes. Section 5.2.1 re-introduces the concept of node in relation to the node-model described above and how it interacts with the rest of the SOS. Section 5.2.2 describes the services provided by the node and their interaction with the rest of the SOS.

5.2.1 The Node

A SOS node is a physical device that belongs to the SOS network[93] and is identified by a GUID, has a pair of keys to digitally sign/verify network requests, and provides a set of services to manage the SOS model (see Section 5.1.3 for the node model). This section, and the ones to follow, will address how a node works, what services it provides and how it interacts with the rest of the *Sea of Stuff*.

5.2.1.1 Node Requests

The pair of keys associated with a node serves two functions: generating the GUID of the node, as seen in Section 5.1.3, and to signing/verifying requests between nodes.

One of the challenges in network communication is to ensure that requests between nodes are secure from attackers and that nodes can trust the received requests. The SOS provides authentication, non-repudiation, and integrity over nodes' communication by signing outgoing requests. The diagram in Figure 5.23 illustrates how network requests are signed/verified by nodes. In this example, there are two nodes (ABC and DEF), with node ABC making a request to node DEF. When node ABC prepares the request for node DEF, it adds the HTTP header *sos-challenge* to the request, sets its value to a random string, and then signs the request using its private key. The resulting signature is sent along with the request and can be verified by node DEF using the public key of ABC. Similarly, node DEF signs the response to node ABC using its private key. For this mechanism to work, a node should have knowledge of the other node's public key, so that the signatures can be verified.

Signing network requests makes the SOS more resistant to *man-in-the-middle* (MITM) attacks, because the attacker must know the private key to sign and perform valid requests on behalf of someone else. It should also be noted that the random string that needs to be signed should be long enough as to avoid that an attacker can learn what signature corresponds to what random string by eavesdropping, knowledge which is useful for replay attacks. In the current implementation the random string is 1024 bits long.

[93] Locatable via an IP address.

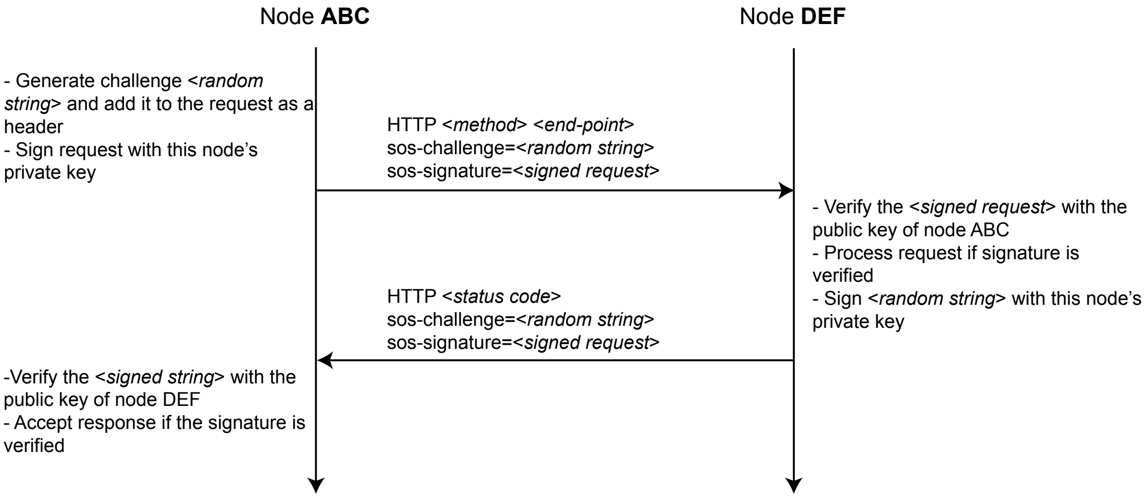

Node **ABC**
Node **DEF**
- Generate challenge <*random string*> and add it to the request as a header
- Sign request with this node's private key
HTTP <*method*> <*end-point*>
sos-challenge=<*random string*>
sos-signature=<*signed request*>
- Verify the <*signed request*> with the public key of node ABC
- Process request if signature is verified
- Sign <*random string*> with this node's private key
HTTP <*status code*>
sos-challenge=<*random string*>
sos-signature=<*signed request*>
-Verify the <*signed string*> with the public key of node DEF
- Accept response if the signature is verified

Figure 5.23: Diagram showing how requests between two nodes are signed.

Further, network requests can be encrypted using a similar pattern. The security of the SOS can also be improved by using HTTPS instead of its unencrypted version HTTP. The current prototype, however, does not support HTTPS requests.

5.2.2 Services

A SOS node provides six services, each designed around a specific aspect of the SOS model:

- **Manifest-Data management service** (MDMS): manages manifests and allows manifests and data stored in the SOS to be found.
- **Node management service** (NMS): allows nodes in the SOS to be discovered.
- **Storage management service** (SMS): manages the storage of atoms.
- **Metadata management service** (MMS): provides a metadata extraction interface and abstracts the location of metadata.
- **User-Role management service** (URMS): manages users and roles.
- **Context management service** (CMS): manages contexts within the nodes and across the domain and codomain of contexts.

A node's implementation is configurable so that users can decide what services to expose via the REST API and which to keep private. As a result, the SOS network can have highly specialised and optimised nodes for a given service, or nodes that just have access to the SOS but do not provide any service to other nodes. The resulting distributed system is similar to an asymmetric "shared-nothing" file system (see Section 2.2.4.3). An asymmetric system is usually more complex of a standard client-server architecture, however the ability to configure what services a node exposes to the rest of the network gives the SOS architecture the potential to adapt its structure and behaviour based on the state of the network itself. For example, storage nodes in part of a network can be made available dynamically when there is a high-demand for storing content.

5.2.2.1 Manifest-Data Management Service - MDMS

The *manifest-data service* is responsible for the management of all the types of manifests in terms of storage, querying and deletion. All other services interact with the MDMS - locally to the node - whenever:

- A manifest has to be stored.
- A manifest has to be retrieved.
- A manifest has to be deleted.
- All manifests of some given type have to be found.
- All versions of an asset have to be found.
- The tips or head versions of an asset have to be found (operations valid within the local node only).

Code 5.21 describes the MDMS interface using Java code.

```java
void addManifest(Manifest manifest, boolean storeLocally, NodesCollection nodes, int replication);

Manifest getManifest(GUID guid) throws ManifestNotFoundException;
Set<GUID> getManifestsRefs(ManifestType type);
```

```

void delete(GUID guid) throws ManifestNotFoundException;

// Methods valid for Assets and Contexts only.
Set<GUID> getVersions(GUID invariant);
GUID getHead(GUID invariant) throws HeadNotFoundException;
void setHead(GUID invariant, GUID guid);
Set<GUID> getTips(GUID invariant) throws TipNotFoundException;
```

Code 5.21: Basic MDMS API.

```
POST /sos/mdms/manifest                                   # add manifest to node
GET  /sos/mdms/manifest/guid/{guid}                       # get manifest with given GUID
GET  /sos/mdms/manifest/guid/{guid}/challenge/{challenge} # challenge node for possession of manifest with given GUID
DELETE /sos/mdms/manifest/guid/{guid}                     # delete manifest with given GUID
GET  /sos/mdms/version/invariant/{invariant}              # get all the known versions for the given asset
```

REST API 5.22: MDMS REST API.

The operations to add a manifest to and get a manifest from the SOS are explained in more details in the next subsections.

Adding Manifests

The operation of adding a manifest to a node via the MDMS involves two steps: (I) adding the manifest into the node internal storage and (II) replicating it to the rest of the SOS, if specified. Figure 5.24 explains these two steps using a flowchart diagram. There are two important aspects shown in the diagram: a manifest can be stored locally or not and the replication process is an asynchronous task, so that the node can continue operating normally while the task is carried out.

Furthermore, it should be noted that the MDMS has to interact with the *node discovery service* to find the nodes where to replicate the manifest.

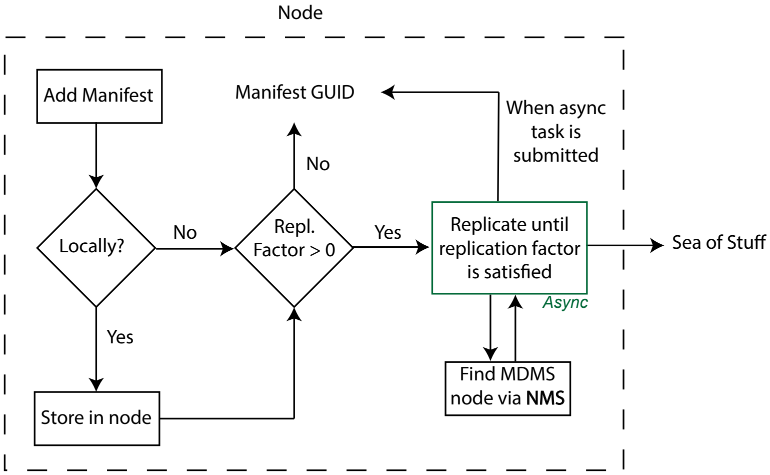


Figure 5.24: Flowchart diagram showing how manifests are added to a node and, optionally, replicated to multiple nodes of the SOS.

Finding and Getting Manifests

Manifests stored locally in the node are directly found by the MDMS through its data structures and retrieved from the node storage. If the manifest is not stored in the local node, other MDMS nodes are interrogated until the manifest is found. In this case, the interaction between the node and the rest of the SOS is synchronous.

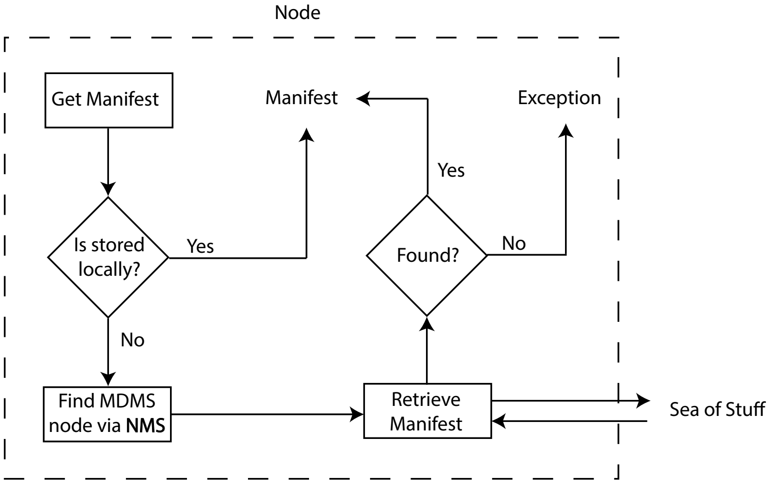


Figure 5.25: Flowchart diagram showing how manifests are found and retrieved from the MDMS.

5.2.2.2 Node Management Service - NMS

The *node discovery service* provides two main functions: (I) enabling a node to be registered into the SOS and (II) allowing nodes in the SOS to be found (see Code 5.23). Nodes are represented as manifests, thus the NMS interacts with the MDMS to store and retrieve all the node manifests. The NMS provides the ability to find nodes in the SOS by their GUIDs or based on the services they provide. To provide a uniform behaviour, the NMS provides these features by querying the MDMS for the matching node manifests.

```
void registerNode(Node node, boolean localOnly) throws NodeRegistrationException;
Node getNodeManifest(GUID guid) throws NodeNotFoundException;
Set<GUID> getNodes(NodeType type);

deleteNode(GUID guid) throws NodeNotFoundException;
```

Code 5.23: Basic NMS API.

```
POST /sos/nms/node              # register node (passed in the body of the request as JSON)
GET  /sos/nms/node/guid/{guid}  # get node manifest with given GUID
GET  /sos/nms/service/{service} # get nodes that match type of service
DELETE /sos/nms/node/guid/{guid} # deletes the node with the given GUID
```

REST API 5.24: NMS REST API.

Joining the SOS

A node joins the SOS by registering itself with a known NMS node. When the node is started, it contacts its known bootstrap nodes, which make the node known to the rest of the SOS.

Finding nodes

A node is found through the NMS service by interrogating the local node and the known NMS nodes, via the `getNode(GUID guid)` method (see Code 5.23). Other NMS nodes are queried through the HTTP call `GET /sos/nms/node/guid/guid` (see REST API 5.24). In the reference implementation presented in Chapter 6, nodes are found by interrogating the already-known nodes, but a DHT or more advanced techniques could also be implemented.

5.2.2.3 Storage Management Service - SMS

The SMS provides an abstraction over the storage of atoms through four main functions (see Code 5.25 for the SMS API):

- Adding atoms to the SOS.
- Getting atoms from the SOS.
- Challenging the local node to prove its possession of a given atom.
- Deleting atoms from the SOS.

```java
AtomManifest addAtom(AtomBuilder atomBuilder) throws StorageException; // The AtomBuilder object contains the information about the atom to be added

Atom getAtomContent(AtomManifest atomManifest) throws AtomNotFoundException;
Atom getSecureAtomContent(SecureAtomManifest atomManifest, Role role) throws AtomNotFoundException;

GUID challenge(GUID guid, String challenge);

void deleteAtom(GUID guid) throws AtomNotFoundException;
```

Code 5.25: Basic SMS API.

```
POST /sos/sms/atom                                 # add atom to storage node
GET  /sos/sms/atom/guid/{guid}                     # get atom with given GUID
GET  /sos/sms/atom/guid/{guid}/protected           # get protected atom with given GUID
GET  /sos/sms/atom/guid/{guid}/challenge/{challenge} # challenge node for possession of atom with given GUID
DELETE /sos/sms/atom/guid/{guid}                   # delete atom with given GUID
```

REST API 5.26: SMS REST API.

Add Atoms

Atoms are added to a node and to remote storage nodes through the SMS. The process of adding atoms to a node and to the rest of the SOS is similar to the one for adding manifests. However, as illustrated by the flowchart in Figure 5.26, the atom can also be encrypted using the approach described in Section 5.1.6.3. An atom manifest is created after the atom is stored into the node and added to the MDMS. Data replication is performed asynchronously over a set of nodes (*e.g.*, the codomain of a context) and the atom manifest is updated accordingly with the new locations.

Alternatively, the atom can be replicated once to a storage node, letting the remote node perform the rest of the replication over the specified nodes until the replication factor is satisfied.

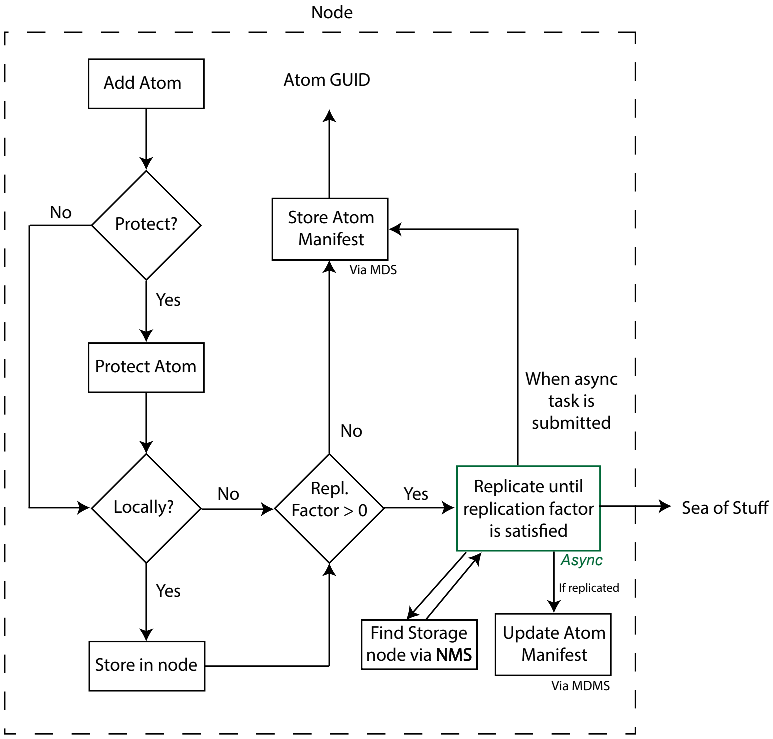


Figure 5.26: Flowchart diagram showing how atoms are added to the SMS and, optionally, replicated to other nodes of the SOS. It should be noted how the SMS interacts with the NMS to find the nodes where to replicate the atom and the MDMS to add/update the atom manifest.

Get Atoms

Atoms are retrieved directly from the locations stored in the atom manifest. If none of the locations returns the correct data, other nodes — previously known or via the NMS — in the network in the SOS providing storage service are contacted. Retrieving protected atoms, however, requires a few additional steps. First of all, the user requesting the atom must specify the role to use to decrypt the data. Then, the atom is decrypted and returned to the user as explained in Section 5.1.6.3.

Proof of Atom Possession

An important property for nodes of a distributed system is the ability to prove their possession of data without necessarily having to send the data to the node requesting such proof. In the SOS, for example, contexts can have *check policy* functions to ensure that atoms are replicated at least N-times. Clearly, a node can retrieve all the replicas and verify their integrity. However, this approach is expensive and adds unnecessary stress to the SOS network. In alternative, a node can challenge the nodes storing the replicas using the approach below (see also Figure 5.27):

1. The local node gets the atom in question and temporarily appends a random string, the challenge, to it.
2. A hash of the temporary data is calculated, h'.
3. The local node challenges the node that might have the replica, for the given atom's GUID and the challenge string.
4. The remote node gets the atom, appends the challenge and calculates the hash of the temporary new data, h''.
5. The remote node sends h'' back to the local node.
6. There are two possible outcomes:
    - **Case 1**: h'' matches h', therefore the remote node has proved possession of the atom.
    - **Case 2**: h'' does not match h', therefore the remote node was unable to prove possession of the atom.

The principle behind this technique is that only nodes that have the atom can properly generate h'' after appending the challenge string to it. Long randomly generated challenge

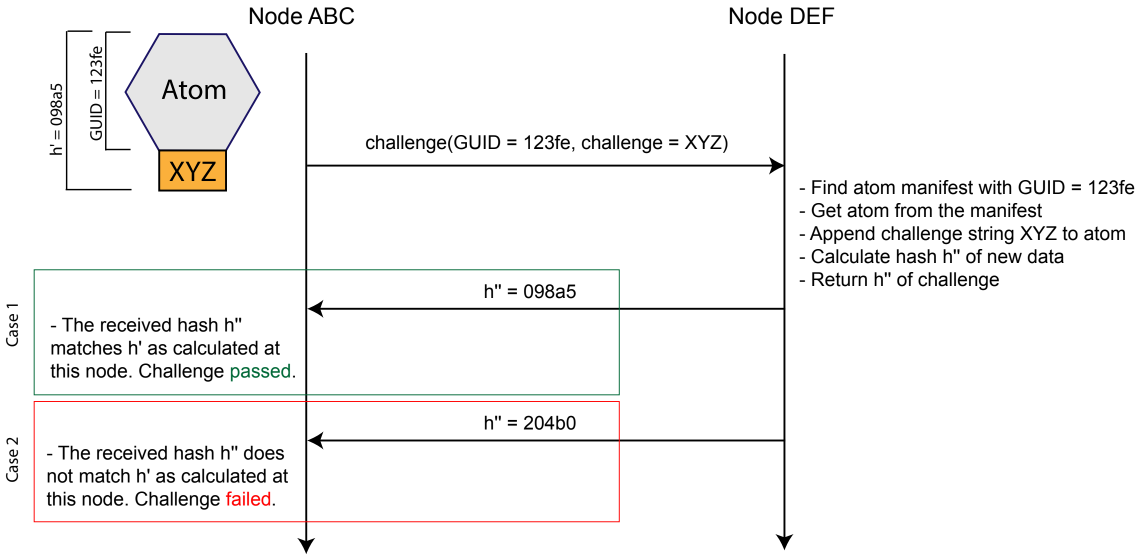

Node ABC
Node DEF
h' = 098a5
GUID = 123fe
Atom
XYZ
challenge(GUID = 123fe, challenge = XYZ)
- Find atom manifest with GUID = 123fe
- Get atom from the manifest
- Append challenge string XYZ to atom
- Calculate hash h" of new data
- Return h" of challenge
Case 1
h" = 098a5
- The received hash h" matches h' as calculated at this node. Challenge passed.
Case 2
h" = 204b0
- The received hash h" does not match h' as calculated at this node. Challenge failed.

Figure 5.27: Diagram showing a node, ABC, requesting proof of possession for the atom *123fe* to node DEF.

strings should be used as a protection against replay attacks.

Finally, the MDMS can be extended to use the same approach to prove the **possession of a manifest** by appending a challenge string to the end of a manifest (represented as a JSON document).

5.2.2.4 Metadata Management Service - MMS

The *metadata management service* is the component of the node responsible for extracting metadata from atoms and/or compounds and storing metadata manifests (see Code 5.27). The methods to add and retrieve the metadata to and from the MMS are simply abstractions over the MDMS. The complexity of the metadata extraction process depends on the amount and precision of the metadata to be extracted. The ability to have configurable nodes allows the SOS to have highly specialised MMS nodes used solely for the extraction and storage of the metadata. This feature is particularly useful in a system where some devices have limited capabilities, such as smartphones, and the expensive computation is moved to some more powerful servers.

```
void addMetadata(Metadata metadata);
Metadata getMetadataManifest(GUID guid) throws MetadataNotFoundException;

```

```
// Extracts metadata from some content identified by a GUID
Metadata processMetadata(GUID guid) throws MetadataException;

void deleteMetadata(GUID guid) throws MetadataNotFoundException;
```

Code 5.27: Basic MMS API.

```
POST /sos/mms/metadata                 # add metadata to MMS node
GET  /sos/mms/metadata/guid/{guid}     # get metadata with given GUID
POST /sos/mms/metadata/process         # process metadata at MMS node
DELETE /sos/mms/metadata/guid/{guid}   # delete metadata with given GUID
```

REST API 5.28: MMS REST API.

5.2.2.5 User-Role Management Service - URMS

The *user-role service* handles users and roles within the SOS. The URMS provides the basic functionalities to create, add, and retrieve users and roles to and from the node. An important aspect of the URMS is also the management of the private-public keys associated with the users and the roles. Moreover, the URMS provides utility functions to enable one role to *grant access* on a given SOS entity to another role (see Section 5.1.6.3).

```
void addUser(User user);
User getUserManifest(GUID userGUID) throws UserNotFoundException;
Set<GUID> getUsers();

void addRole(Role role);
Role getRoleManifest(GUID roleGUID) throws RoleNotFoundException;
Set<GUID> getRoles(GUID userGUID) throws RoleNotFoundException;

deleteUserRole(GUID guid) throws EntityNotFoundException; // Deletes the user/role matching the GUID

grantAccess(GUID protectedEntity, Role granterRole, Role granteeRole) throws EntityNotFoundException;
```

Code 5.29: Basic URMS API.

```
GET  /sos/urms/guid/{guid}        # get user/role with given GUID
GET  /sos/urms/user/{guid}/roles  # get known roles of given user
POST /sos/urms/user               # add user to URMS node
POST /sos/urms/role             # add role to URMS node
DELETE /sos/urms/guid/{guid}    # delete user/role with given GUID
```

REST API 5.30: URMS REST API.

5.2.2.6 Context Management Service - CMS

The *context management service* manages the contexts and its computational work units in terms of storage and scheduling. The storage management of the context, predicate, and policy manifests relies on the MDMS. Additionally, the CMS is responsible for compiling the predicate and policy code, embedded in their respective manifests. The CMS schedules all the predicate, policy-apply and policy-check tasks separately as described by the context life cycle.

```
GUID addContext(ContextBuilder contextBuilder);

Context getContextManifest(GUID guid) throws ContextNotFoundException;
Set<GUID> getContents(Context context);
Set<GUID> getContexts();

void deleteContext(GUID invariant) throws ContextNotFoundException;
```

Code 5.31: Basic CMS API.

```
POST /sos/cms/context                        # add context to CMS node
GET  /sos/cms/context/guid/{guid}            # get context with given GUID
GET  /sos/cms/context/guid/{guid}/contents # get contents for given context
DELETE /sos/cms/context/invariant/{guid}  # delete context with given GUID
```

REST API 5.32: CMS REST API.

The context and its computational work units are distributed over the context's domain and run at the respective nodes. If a node within the domain cannot run the context over its assets, then the assets are distributed over all the nodes that can run the context.

5.2.3 Considerations

The SOS defines a model for managing data as well as an architecture of services which understand and control the different aspects of the model. The SOS, however, does not define how nodes and services are organised and whether there is any relationship between nodes and the content they store. Distributed systems can be centralised, decentralised, or a mix of the two. In the SOS, as presented in this chapter, nodes are managed as in an unstructured P2P network and both nodes and content are distributed and found greedily. The general design of the SOS can be supported by a P2P architecture as well as a client-server - or centralised - architecture.

The scope of this thesis is to present a generic model for managing data in a distributed system. Thus, further discussion about the type of architecture underlying the SOS is left for discussion to Chapter 7 — *Comparative Evaluation* — and Chapter 9 — *Conclusions and Future Work*.

Reference Implementation

The previous chapter presented the design of the *Sea of Stuff* model and its architecture. A prototype of the SOS has been developed using the Java programming language (version 8) and external libraries where needed (*e.g.*, to extract metadata from atoms or to build a RESTful server).

This chapter describes the most important aspects of the reference implementation of the SOS and gives two examples of applications built on top of the SOS. Section 6.1 describes the most important implementation details about the services provided by a SOS node, while Section 6.2 shows how content is stored in the local file system. Section 6.3 briefly illustrates how nodes can be configured to have different behaviours and expose different services. Section 6.4 describes two example applications developed on top of a SOS node. Finally, Section 6.5 discusses the limitations of the prototype.

The prototype developed for this work is available as free software under the *GNU General Public License v3.0* at `https://github.com/sea-of-stuff`

6.1 Services

6.1.1 Storage Management Service

The SMS is responsible for storing, retrieving, and deleting atoms across the SOS (see Section 5.2.2.3). Proof of atom possession is also provided through the SMS.

Storing Atoms in the Sea of Stuff

Storing an atom is a four step process (see Code 6.1):

1. The atom is saved to the *node's storage* using a randomly generated name, since its GUID is not known yet.

2. The GUID of the atom is calculated and the name of the saved atom is updated.[94]

3. The manifest for the atom is created and saved to the node via the MDMS.

4. Optionally, the atom is replicated to other SOS nodes (as specified by context's policies).

The *node's storage* is where atoms, manifests, and all persistent internal data structures are stored (see Section 6.2 for more information about its structure). The prototype uses the *castore* utility library to abstract the node's storage, so that any form of data, from atoms to manifests alike, can be stored in the node without any knowledge of the underlying storage. *Castore* provides a common storage abstraction for the local file system and cloud storage services, such as Amazon S3, Dropbox, and Google Drive (see Appendix C for more information on *castore*).

The SMS is also responsible for replicating atoms to other nodes in the SOS. Atoms are replicated by calling the REST end-point *POST /sos/sms/atom*[95] of a given node.[96] The atom is replicated to as many nodes as specified by the replication factor.[97] The retrieval of atoms from remote nodes is handled similarly by making calls to the appropriate REST end-point.

```
/**
 * Adds an atom to the SOS and returns an AtomManifest.
 * @param AtomBuilder - This is an object containing the information to add the atom.
 */
AtomManifest addAtom(AtomBuilder atomBuilder) throws StorageException {
    Set<LocationBundle> locations = storeAtom(atomBuilder);
    GUID guid = generateGUID(atomBuilder);
```

[94] The implementation presented in this thesis uses the Apache commons-codec library for hashing data. The library is available at the following URL: `https://commons.apache.org/proper/commons-codec/` [last accessed on 30/04/2018].

[95] The atom's bytes are sent as part of the request's body.

[96] The node where the atom is replicated must be configured so that its SMS is exposed via REST.

[97] The replication factor constraint may not be satisfied if some of the nodes are unavailable.

```
    renameAtom(guid, locations);

    AtomManifest atomManifest = ManifestFactory.createAtomManifest(guid, locations, atomBuilder);
    manifestsDataService.addManifest(atomManifest);

    int replicationFactor = atomBuilder.getReplicationFactor();
    if (replicationFactor > 0) {
        Task atomReplicationTask = new AtomReplicationTask(atomManifest,
               atomBuilder.getCodomain(), // List of nodes where the atom can be replicated
               replicationFactor,
               this, // SMS, needed to update the manifest with replica locations
               nodeDiscoveryService); // Needed to be able to contact the replica nodes
        TasksQueue.instance().performAsyncTask(atomReplicationTask);
    }

    return atomManifest;
}
```

Code 6.1: Java code for the **add atom** operation of the SMS.

Retrieving Atoms in the Sea of Stuff

In the current SOS prototype, the retrieval of atoms relies on the locations stored within the atom manifest and any locations that the MDMS has associated with the atom to be retrieved. In addition, locations are processed in a specific order, so that the local node is interrogated before any remote location. The remote locations are ordered based on their type (*i.e.*, cache, persistent, external).

```
Atom getAtom(IGUID guid) throws AtomNotFoundException {
    Manifest manifest = mdms.getManifest(guid);
    if (manifest.getType() == ManifestType.ATOM) {
        Atom atom = (Atom) manifest;

        // Get known locations about atom, which can be stored in the atom or cached in the MDMS.
        // Locations are ordered as follows:
        // - Local node
        // - Marked as cache
        // - Marked as persisted
        // - Marked as external
        Set<Location> locations = mdms.getLocations(atom);
        for(Location location:locations) {
            Atom atom;
            try {
                atom = location.getAtom();
            } catch(AtomNotFoundException e) {
                continue; // Try next location
            }
            return atom;
        }
    }

    throw new AtomNotFoundException(guid);
}
```

Code 6.2: Java code for the **get atom** operation of the SMS.

Deleting Atoms in the Sea of Stuff

When deleting atoms from the SOS, the SMS iterates over all the known locations for the atom and issues a deletion request for each of them. The global deletion of an atom (as well as manifests) cannot be guaranteed because it is not possible to verify that a remote node has actually deleted it. Further, global deletion cannot be guaranteed because keeping track of all the replicas of an atom is infeasible.

Proof of Atom Possession

Proof of atom possession is implemented following the algorithm described in Section 5.2.2.3[*Proof of Atom Possession*].

6.1.2 Manifest-Data Management Service

The implementation of the MDMS is based on a three layer manifest storage abstraction (see Figure 6.1):

- **Cache layer**. Stores a limited number of manifests in-memory (8192 by default), providing fast queries and read access for manifests. The cache implements a least recently used (LRU) policy, so that recently used manifests are faster to access. A LRU cache improves access over content that has been accessed recently. Potentially, other caching strategies could be used if one wishes to optimise the system for different use cases.[98]

- **Local layer**. Stores the manifests to the local *node's storage*, similarly to how atoms are stored by the SMS. If a manifest is found, then the *cache* layer is updated accordingly.

- **Remote layer**. This is an abstraction over all other nodes of the SOS. If a manifest is found, then the *cache* and *local* layers are updated accordingly.

[98] Note that this is a cache in the traditional sense, unlike the *cache* hint used within an atom manifest, which is used solely to highlight locations that expose better data access properties.

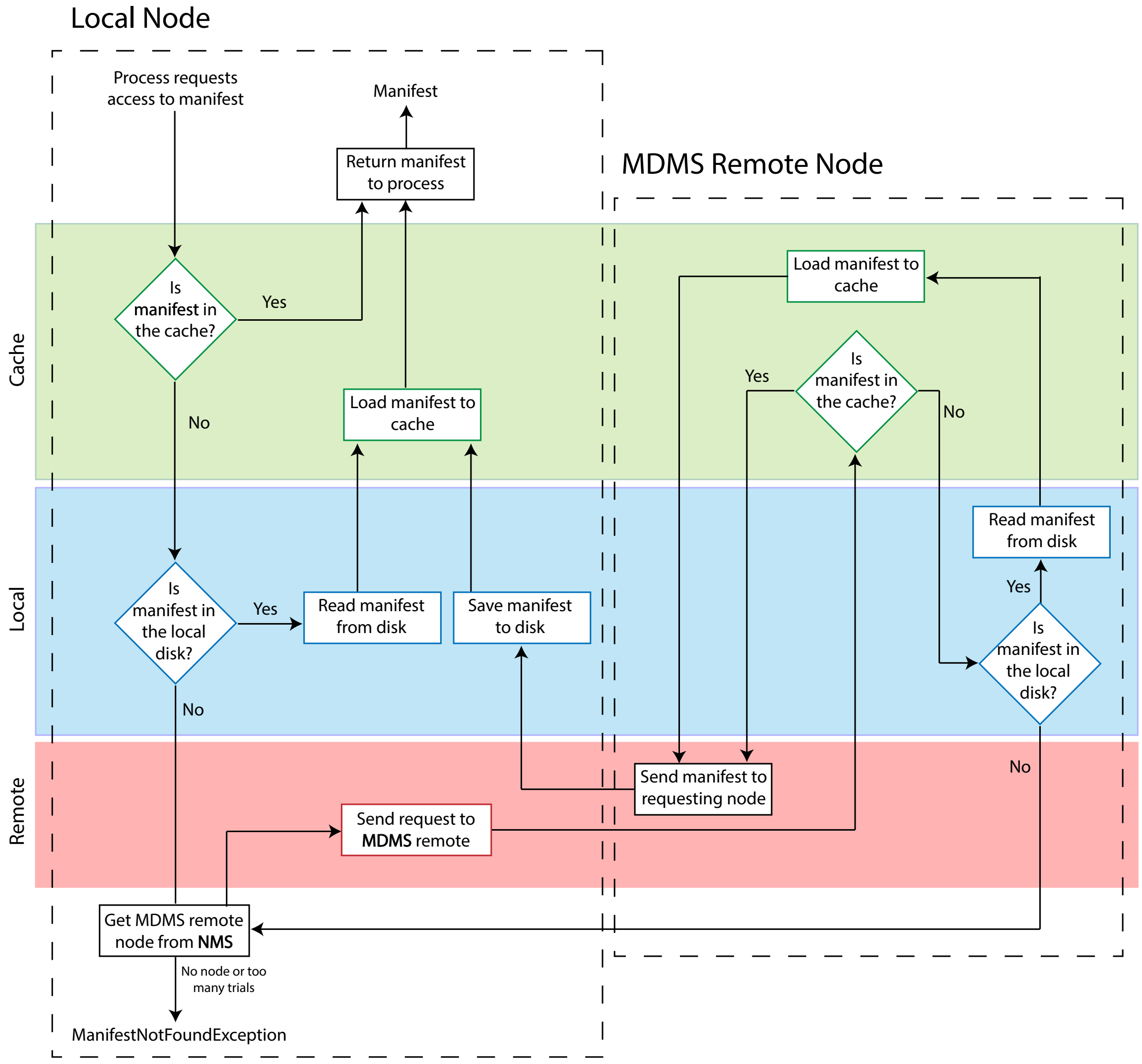

Local Node
Process requests access to manifest
Manifest
Return manifest to process
MDMS Remote Node
Cache
Is manifest in the cache?
Yes
No
Load manifest to cache
Load manifest to cache
Yes
Is manifest in the cache?
No
Local
Is manifest in the local disk?
Yes
Read manifest from disk
Save manifest to disk
No
Read manifest from disk
Yes
Is manifest in the local disk?
No
Remote
Send manifest to requesting node
Send request to MDMS remote
Get MDMS remote node from NMS
No node or too many trials
ManifestNotFoundException

Figure 6.1: Flowchart diagram showing the interaction between the three storage layers of the MDMS for the **add manifest** operation: the **cache** layer is interrogated first, then the **local** layer is checked, and finally the node interacts with the **remote** layer as last resort.

6.1.3 Metadata Management Service

The MMS provides two types of services: (I) the storage and management of the metadata model and (II) the extraction of metadata from some given data. The first service is provided by managing the metadata manifests through the MDMS. The extraction of the metadata is achieved by processing the data using the Apache Tika™ toolkit[99], which provides a set of tools to detect and extract metadata and text from data of different type (*e.g.*, doc, docx, pdf, png, and jpeg).

6.1.4 Node Management Service

The implementation of the NMS consists of a basic thin wrapper around the MDMS. The NMS also provides a set of utility methods to find nodes based on the services provided. For example, the NMS can be queried to return the known nodes that provide a storage services or a metadata service. Node filtering is particularly useful when other services need to interact with the SOS through the NMS.

6.1.5 User-Role Management Service

The URMS service, like the MMS and the NMS services, is implemented as a thin wrapper around the MDMS. In addition, the URMS manages the keys associated with users and roles, which are all stored in the *node's storage* (see Section 6.2).

6.1.6 Context Management Service

The CMS provides two main functions: managing the context, predicate, and policy manifests and scheduling the computation associated with the context. The management of the context and its components is handled through the MDMS. The previous chapter described the *context life cycle* (see Section 5.1.7.5), how the computational units are scheduled periodically, and how contexts change over time. The reference implementation presented in this chapter schedules the computational units using a Java *ScheduledExecutorService*, which allows tasks to be run periodically. The configuration of the node defines how often the computational units are run. The CMS also schedules tasks for distributing

[99] Apache Tika. `https://tika.apache.org/` [last accessed on 27/03/2018].

contexts over their domains and synchronising the contexts' contents among nodes of the domain.

6.1.7 Node Maintenance

The node maintenance mechanism is responsible for controlling how much data and how many manifests are stored in a node and for removing any data or manifest that is replicated a sufficient number of times and/or marked as *cache* (as described in Section 5.1.4.1[*Atom Manifest*]). The maximum amount of data/manifests that can be stored can be set in the node's configuration.

6.1.8 REST API

The services provided by the node are exposed to the other nodes of the SOS via HTTP, with the functions described by a REST API, as described in Section 5.2.2 of the previous chapter. The REST API is built using Jersey[100], a framework that implements JAX-RS, and runs on top of Jetty[101], a lightweight web server and servlet engine. The mapping between the REST API and the methods provided by its services is 1:1, so the REST API is simply a thin wrapper around the node services.

6.1.9 Instrumentation

The implementation of the SOS presented in this work has been instrumented to record user-defined metadata (*e.g.*, time to run a particular method, number of iterations of a particular loop, *etc.*). The instrumentation of the node is enabled for the experiments described in Chapter 8. Code 6.3 shows an example of code instrumentation for the *runPredicatesPeriodic()* method, where the instrument object records the overall time it takes to run the predicate for all the managed contexts.

```
void runPredicatesPeriodic() {
    ThreadSettings predicateThreadSettings = getPredicateThreadSettings();
    service.scheduleWithFixedDelay(() -> {

        long start = System.currentTimeMillis();
```

[100] Jersey. RESTful Web Services in Java. https://jersey.github.io/ [last access on 23/02/2018].

[101] Jetty. https://eclipse.org/jetty/ [last access on 23/02/2018].

```
        for (Context context : getContexts()) {
            runPredicate(context)
        }
        long end = System.currentTimeMillis();
        Instrument.instance().measure(StatsTYPE.thread, StatsTYPE.predicate, "Thread_Predicate",
             start, end);

    }, predicateThreadSettings.getInitialDelay(), predicateThreadSettings.getPeriod(),
         TimeUnit.SECONDS);
}
```

Code 6.3: Java code describing the instrumentation of the *runPredicatesPeriodic()* method.

The code of the presented prototype is instrumented for multiple services. The interested reader can inspect the source code of the SOS.[102]

[102] SOS - Sea of Stuff. Source code available at `https://github.com/sea-of-stuff` [last accessed on 30/03/2018].

6.2 Node Storage

The entities of the SOS model and the persistent internal data structures of the node are stored in the node's internal storage, which is organised as follows:

- **manifests/** This is the location where all the manifests are stored. A manifest is stored as a JSON document and named using its GUID.
- **atoms/** This is the location where all the atoms are stored. Atoms are named using their GUIDs.
- **keys/** This is the location where the keys of the users and roles are stored. The digital signature algorithm used is *SHA256* with *RSA* and key length of 512 bits. The asymmetric encryption algorithm used is *RSA* with a key length of 2048 bits. Keys are stored using the following file naming scheme:
    - *⟨User/Role GUID⟩***.key** – private key for the digital signature of users and roles.
    - *⟨User/Role GUID⟩***.crt** – public key for the digital signature of users and roles.
    - *⟨User/Role GUID⟩***.pem** – private key for asymmetric encryption as used by the roles.
    - *⟨User/Role GUID⟩***_pub.pem** – public key for asymmetric encryption as used by the roles.
- **node/** This is the location where all the node internal databases, indices and caches are stored as well as the node's private and public keys.
- **java/** This is the location where the compiled computational units are stored as *⟨name⟩.class* files.

Figure 6.2 illustrates a typical example of node structure on the file system.

```
sos
├─ atoms
│  ├─ SHA256_16_25a366a90726c4f2aae072554552a5c606f4a0462982349f1e2fed8e0cc7c29b
│  ├─ SHA256_16_66feef39d5e4dddbbc211f6ce8f517be4a49859f68b4f589e67f73e223e8c86a
│  └─ ...
├─ java
│  └─ uk/ac/standrews/cs/sos/impl/context
│     ├─ SHA256_16_2230581b4234bdfe697b8707bd9a936063c33ccbbcdf6bf3f606136204f072db.class
│     └─ ...
├─ keys
│  ├─ SHA256_16_2b4cd97d702dbcfcaff11e84bb7ba47e8fda9c9849235d68bb03bf8ef86cecd2.crt
│  ├─ SHA256_16_2b4cd97d702dbcfcaff11e84bb7ba47e8fda9c9849235d68bb03bf8ef86cecd2.key
│  ├─ SHA256_16_2b4cd97d702dbcfcaff11e84bb7ba47e8fda9c9849235d68bb03bf8ef86cecd2.pem
│  ├─ SHA256_16_2b4cd97d702dbcfcaff11e84bb7ba47e8fda9c9849235d68bb03bf8ef86cecd2_pub.pem
│  └─ ...
├─ manifests
│  ├─ SHA256_16_01c748f22eade8555da2d4197e6394825b8affddf7559e7ebeebd42579220ccf
│  ├─ SHA256_16_0fa544753e238fee6676b3141667b07f5067257903f3d937c82f6151ba942947
│  ├─ SHA256_16_2230581b4234bdfe697b8707bd9a936063c33ccbbcdf6bf3f606136204f072db
│  ├─ SHA256_16_25a366a90726c4f2aae072554552a5c606f4a0462982349f1e2fed8e0cc7c29b
│  └─ ...
└─ node
   ├─ id_rsa.crt
   ├─ id_rsa.key
   ├─ node.db
   ├─ cms.index
   ├─ locations.index
   ├─ manifests.index
   ├─ manifests.cache
   ├─ mdms.index
   └─ urms.index
```

Figure 6.2: Example of folder structure for the node internals.

6.3 Node Configuration

The node's settings can be specified through a configuration file that is processed when the node is started. The main settings that can be controlled through the configuration file are:

- Node hostname and port for the REST service.
- Settings for each service (*e.g.*, NMS, MDMS, SMS, *etc.*).
- Internal store type (*e.g.* the local file system, Amazon S3, Dropbox, *etc.*) and path.
- Bootstrap nodes: list of already known nodes that are used to join the SOS. Nodes are described in terms of their GUIDs, addresses, and public keys.

More information about the node configuration can be found in the Appendix A.1.

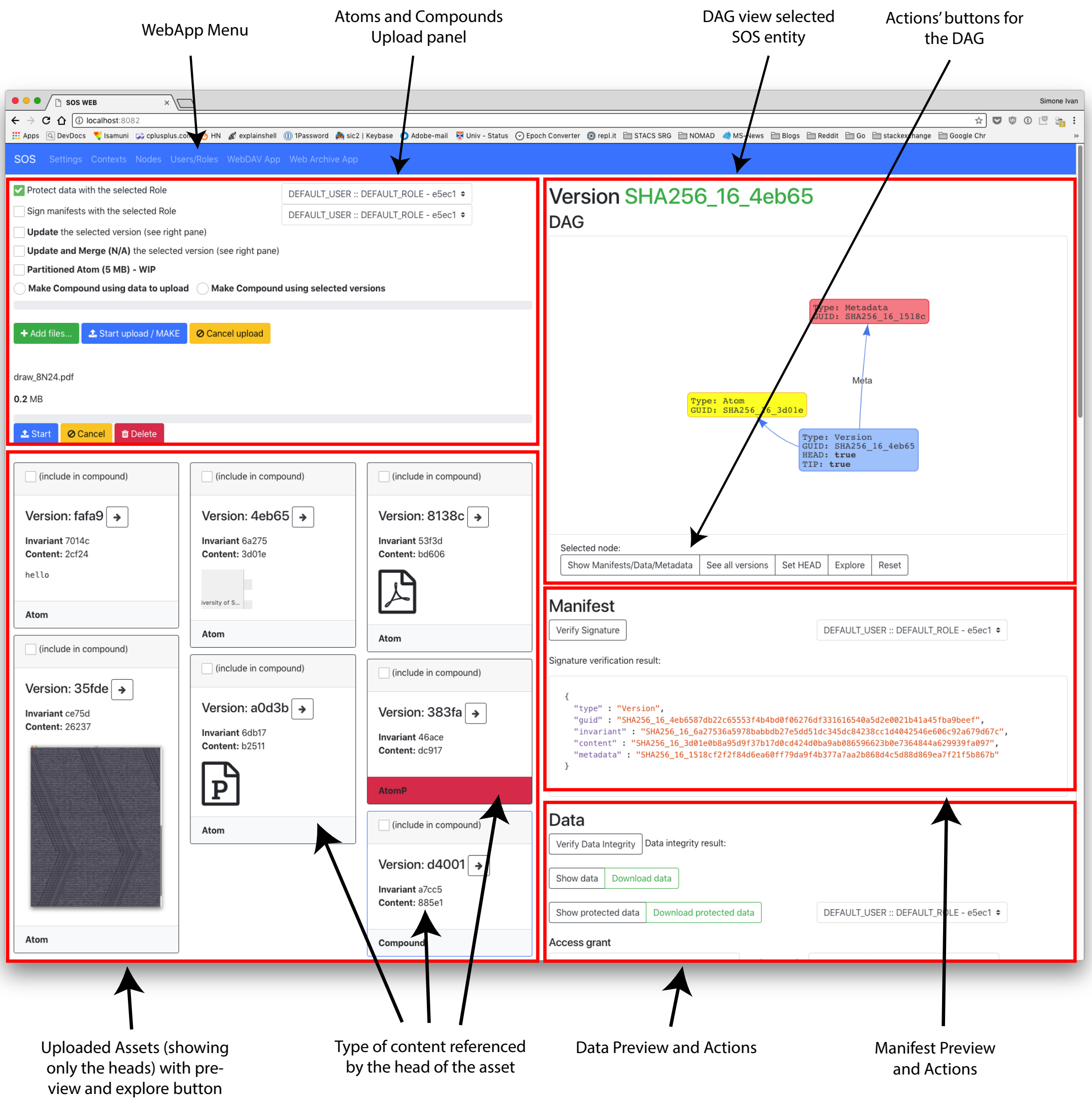
6.4 Example Applications

6.4.1 WebApp

The *WebApp* enables its users to understand the SOS model and interact with the node and the SOS via the browser. The WebApp allows users to explore the underlying SOS model, understand the relationship between the different entities of the model thanks to visual aids, look at the manifest of each entity, or set the head of an asset. In particular, the WebApp provides the following functionalities:

- **Data management**. Users are able to add and update assets in the SOS, either as atoms or compounds. Figure 6.3 shows an example of how data is managed and inspected via the WebApp.
- **Users and Roles management**. The WebApp's users are able to create SOS users and roles with arbitrary names.
- **Context management**. Users are able to create contexts, inspect their contents over time, and monitor their scheduling.

- **Node management**. Users can monitor the local node and the known remote nodes. This feature is useful for demonstration purposes only.



WebApp Menu

Atoms and Compounds Upload panel

DAG view selected SOS entity

Actions' buttons for the DAG

Uploaded Assets (showing only the heads) with preview and explore button

Type of content referenced by the head of the asset

Data Preview and Actions

Manifest Preview and Actions

```
{
  "type" : "Version",
  "guid" : "SHA256_16_4eb6587db22c65553f4b4bd0f06276df331616540a5d2e0021b41a45fba9beef",
  "invariant" : "SHA256_16_6a27536a5978babbdb27e5dd51dc345dc84238cc1d4042546e606c92a679d67c",
  "content" : "SHA256_16_3d01e0b8a95d9f37b17d0cd424d0ba9ab086596623b0e7364844a629939fa097",
  "metadata" : "SHA256_16_1518cf2f2f84d6ea60ff79da9f4b377a7aa2b868d4c5d88d869ea7f21f5b867b"
}
```

Figure 6.3: Example view of the home page of the WebApp. The home page is divided in two main parts: the left component allows used to add content to the SOS and view the added content, while the right side allows users to explore how content is modelled and stored in the SOS.

6.4.1.1 Limitations

The WebApp is an application built on top of the SOS that allows users to interact with the SOS and understand part of its model. However, the WebApp presents a number of limitations that are listed here:

- The UI/UX is not optimal, since the scope of the WebApp is simply the one to demonstrate some of the features of the SOS.

- The deletion of assets, contexts, users, and roles is not supported.

- Contexts cannot be made inactive (see Section 5.1.7.5[*Context Life Cycle*]).

- The WebApp does not provide any search features. Future work on this direction involves the building of a search engine based on the metadata model. The context model, on the other hand, provides a mechanism to classify content based on its metadata.

- The WebApp instance is accessible to anyone who has knowledge of the IP address of the node where it runs. The current implementation of the WebApp should not be run on production environments and/or without knowledge of its security weaknesses.

6.4.2 SOS-WebDAV Server

The SOS-WebDAV server is another application example built on top of a SOS node. The purpose of the SOS-WebDAV is to allow users to operate on the SOS as they would do on a file system. The Web Distributed Authoring and Versioning (WebDAV) protocol is an extension of HTTP and "allows clients to perform remote web content authoring operations" (RFC 2518 [221]). Similar to the HTTP protocol, WebDAV involves two actors: a server that provides the resources and a client that acts upon the resources stored in the server side. A *resource* is either a file or a directory. The WebDAV protocol, unlike HTTP, provides authoring methods that enable (I) a wider range of operations over the managed resources, such as copying and moving them across locations and collecting them together; and (II) to describe resources using a richer syntax (expressed in XML).

The SOS-WebDAV server is based on the WebDAV server developed as part of the *asa* (Autonomic Storage Architecture) project [222]. The WebDAV server design allows an arbitrary file system to be served under the WebDAV protocol. The file system must implement the interfaces provided by the *fs-sta* project for files and directories.[103]

The main challenge for the integration between the WebDAV server and the SOS is the mapping between the WebDAV resources, files and directories, and the SOS entities of the data model, atoms and compounds. The simplest solution would be to create a single asset that keeps track of all the resources stored on the server side. However, this solution tightly binds a resource and its history to all other resources, and their respective histories, stored in the node. One of the features of the SOS is that everything can be versionable, so a different approach was adopted: each resource is represented as an independent asset that refers either to an atom, for files, or a compound, for directories. Thus, compounds are collections of versions, each representing a particular point in time of an asset, or resource. Section 5.1.4.3[*Versions with compounds of versions*] explains this approach in more detail.

6.4.2.1 Limitations

Application layer file systems (ALFS, see Section 2.2.1.6), such as the ones built on top of FUSE, have reasonably good IO performance, compared to system level file systems. However, WebDAV is widely supported across many operating systems, such as MacOS, iOS, Android, and Windows, while ALFS libraries, such as FUSE, require software installation on the client and *ad-hoc* implementations for each OS. Additionally, implementing an ALFS tends to be complex and error-prone. On the other hand, an ALFS can provide much better performance over a WebDAV service and does not require a running server, which exposes a node to potential security risks.

The WebDAV server implementation presents some bugs which could not be solved within the time-frame of this work. The most important bug of the WebDAV server is its inability to handle large files.

Finally, the SOS-WebDAV support for the functionality provided by the SOS is limited by the WebDAV protocol itself, which reflects the common file system metaphor. End-users should use the SOS-WebDAV application together with the WebApp, in order to have better control of the data, since not all operations supported by the SOS are available

[103] The Appendix C describes the WebDAV and *fs-sta* projects in more details.

via the file system supported by the operating system.

6.5 Overall Prototype Limitations

The prototype presented in this work presents the following limitations and security issues:

- Small files occupy an entire block in the file system. This is very inefficient. A packing operation (similar to git) could be adopted.
- The SOS protocol is built on top of HTTP, which is an insecure protocol. As part of future work, it is suggested to adopt HTTPS plus exploring the possibility to encrypt requests using the public key of the contacted node.
- Predicate and policy functions are written using a set of *ad-hoc* functions provided by SOS prototype. Future work should focus on providing a better definition of the language for the context's computational units.
- The MMS relies on Apache Tika, so the richness and accuracy of the metadata associated to the assets is strictly dependent on the Tika component.
- Context synchronisation works under the assumption that different context instances will eventually be consistent. However, no mechanism for resolving conflicts has been built.
- Requests between nodes are not properly tracked, thus it is not possible to handle circular requests between them. A circular request can happen when one node A asks for some information to node B and node B, in exchange, asks for the same information to node A. Circular requests are transitive (*i.e.*, can involve one or more intermediate nodes) and lead to a deadlock of the parts of the system involved.

In addition, the applications presented in this chapter have been designed and built for demonstrational purposes only. The limitations outlined in the respective sections should be taken into account by anyone wishing to work on this project in the future.

Comparative Evaluation

Storage systems, as seen in previous chapters, are characterised by a significant number of properties, from how data is represented to how it could be versioned or protected. Evaluating all the properties of the **Sea of Stuff** experimentally against other systems is unfeasible due to the large number of properties to take into account and the different nature of the implementations of the systems.

This chapter evaluates the design of the SOS against the storage systems presented in the *Background* and *Literature Review* chapters. The next chapter — *Experimental Evaluation* — evaluates experimentally some specific aspects of the SOS, with a particular focus on the context model.

The scope of the evaluation presented in this chapter is defined by the Hypothesis 1 (**H1**, see Section 1.3), the types of systems presented in the *Background* and *Literature Review* chapters, and the design of the SOS. One of the main goals of the SOS is to re-think the fundamental storage abstractions, such as the file and the directory. These abstractions are evaluated comparatively throughout this chapter against the relevant storage systems.

This chapter is structured similarly to the *Literature Review* chapter, so that the SOS can be compared against each class of storage systems. However, given the large number of storage systems, only a selection of them for each storage type is chosen based on their historical and/or current significance in the state of the art.

7.1 Evaluation of Requirements

This section briefly evaluates the *Sea of Stuff* design and reference implementation, presented in the previous chapters, against the requirements reported in Chapter 4.

7.1.1 End-User Requirements

✓ **EU-1 Location abstraction**: data should be accessible irrespective of the location where it is stored.

The SOS models data as content-addressable, thus de-coupling content and its location.

✓ **EU-2 Accessibility abstraction**: data should be accessible irrespective of the locations from which it is accessed.

Accessibility abstraction is achieved together with location abstraction due to the content-addressable nature of the model.

✓ **EU-3 Users and Roles**: a user (*e.g.*, Simone, Giulia, Al, Graham) should be associated to multiple roles (*e.g.*, home, work, hobby, holiday, photography) and have a different view of the SOS based on their current role.

The SOS models users and roles, which can be used to sign and protect entities of the model or to simply define an ownership relationship. Moreover, users and roles can be used to provide *ad-hoc* views of the SOS via contexts.

✓ **EU-4 Data Protection**: users should be able to enforce arbitrary levels of data protections in terms of what to protect and what encryption algorithms to use.

In the SOS, data can be protected at arbitrary levels of granularity through roles. The quality of protection depends on the length and type of keys used.

✓ **EU-5 Versioning**: it should be possible to version data and metadata at different granularities.

The SOS allows data and metadata to be versioned at arbitrary levels of granularity through the asset entity.

✓ **EU-6 Automatic data management**: users should be able to define rules to automatically manage data in a distributed environment.

The SOS allows data to be automatically managed in a distributed system through the context entity and the predicate and policy computational work units.

7.1.2 Model Requirements

✓/✗ **M-1 Immutability**: the components or entities of the model should be immutable.

All the entities of the model are immutable, except for the atom manifest. The immutable entities are uniquely identified by a GUID and never change over time. The atom manifest serves as a proxy entity for the atom and consists of a list of locations, for the associated atom, that can change over time. The identifier of the atom manifest is generated by hashing the atom and it is immutable over time.

✓ **M-2 Integrity**: it should be possible to verify the integrity of the content managed through the model.

The GUID associated with each entity is a cryptographically secure hash that can be used to verify its integrity. Changing even one bit of the entity's content results in a new GUID, thus failing the integrity test for the entity.

✓ **M-3 De-duplication**: the model should support data de-duplication.

The SOS allows content de-duplication by storing duplicate entities, which have the same GUID, only once within a given node.

✓ **M-4 Self-describing entities**: entities should be self-describable (see *Glossary of Terms*), so that no additional information is needed to use and process a given entity.

All of the SOS manifests contain the necessary information about the entity they describe so that no further information is needed.

✓ **M-5 Independent entities:** it should be possible to distribute content independently of any other modelled entity.

The SOS allows entities to be distributed independently of one another.

✓ **M-6 Computational work:** it should be able to define computational work that runs over the stored content. It should be possible to distribute the computation across the system.

Computational work is modelled through the context model and the predicate and policy entities, which are used by the context to automatically manage content.

7.1.3 Architecture Requirements

✓ **A-1 CAP**: the system should be AP compliant (see Section 2.1.5), so that data is always available, assuming that it is reachable, and that the system is tolerant to network partitions.

Data is always available as long as the nodes storing it are reachable. Tolerance to network failure is in place for managing data as well as computation, via contexts.

✓ **A-2 Failure**: the system should be able to cope with failures using an *eventually consistent model*.

The SOS handles failures by replicating content via *ad-hoc* contexts. A context that is run over a set of nodes is synchronised with an eventually consistent model.

✓ **A-3 Heterogeneity**: the system should work independently of the type of hardware and operating system it runs on.

The SOS model and architecture design are hardware and software independent. Moreover, the reference implementation presented in this thesis has been developed in Java and can be run over any machine supporting a JVM (with JDK version greater or equal than 1.8).

✓/✗ **A-4 Horizontal scalability**: the system should be able to scale horizontally (*i.e.*, adding nodes to the system).

The SOS model could scale horizontally if it were to use a DHT to manage its content. The implementation of a DHT for the SOS, however, was not the focus of this work.

✓ **A-5 Node configuration**: it should be possible to configure a node in terms of what services it provides to the system.

The reference implementation of the SOS presented in the previous chapter allows nodes to configure what services to expose to the rest of the SOS.

✓ **A-6 Decentralisation**: the system should be able to work without the need of a central authority.

The SOS architecture does not rely on a central authority. A SOS node may depend on other nodes that provide node discovery or storage services, but these are decentralised services themselves (*i.e.*, multiple nodes can provide such services).

✓ **A-7 RESTful API**: a node should expose its services via a RESTful API.

The design architecture of the SOS provides the description of a RESTful API, which has fully been implemented in the reference implementation.

7.2 File Systems

7.2.1 The File Metaphor

The file is the primary entity of file systems. As explained in Chapter 2, the file consists of data, attributes, and operations. However, the file abstraction is not always well understood, in particular when projected against the requirements of today [14]. The initial file abstraction was defined around the CRUD (create, read, update, delete) storage operations. However, users nowadays perform file operations, such as move, duplicate, share, synchronise, and version, which are not always comprehended, especially in a distributed setting, where multiple users and machines interact with each other. The file-control block structure, for instance, has remained mostly unchanged over the last six decades. Over the years, most file systems have provided extended attributes supporting the recording of richer metadata than just the file size, ownership, permissions, and timestamps of last creation, last access, and last modification. The file extended attributes have been implemented in multiple ways, from storing metadata within the file-control block structure to using external data structures (*e.g.*, the resource fork for HFS/HFS+ [129] or the metadata indices for the Be File System [130]). In all of these solutions, however, the metadata is still bound to data.

The SOS does not provide a file metaphor, but its **data model** abstractions - the atom, compound, and asset - can be combined and used to support it. The SOS can model data, exactly like the file abstraction, but it can do it so that its location is transparent (see *Glossary of Terms*) to users. In addition, data integrity and versioning are built-in features of the system. However, the most relevant aspect of the SOS model is that data and its abstract representations are decoupled. This allows users of the SOS to understand data for what it is — a sequence of bits — irrespective of where it is located, aggregated, and versioned. In addition, the SOS models **metadata** as extrinsic to the data, unlike the standard file systems approach. In a distributed system, this allows content to be understood and processed through metadata only, without necessarily having to have a copy of the data.

Another attempt to extend the file metaphor, which is more similar to the SOS model, is file tagging, where files are described by named keys, extrinsic to the file. The pStore file system [135] implements tagging through RDF, thus providing a rich semantic schema to record relationships between files and metadata. Composite data formats, such as DCX [168] and the quFile [209], also extend the file abstraction in a similar fashion to the SOS. Composite data formats are discussed in more detail in the relevant sections of this chapter.

Other file metaphors are discussed and compared against the SOS ones in the relevant sections below.

7.2.2 The Directory Metaphor

The second fundamental abstraction of file systems is the directory. The directory is the metaphor that enables content to be organised in hierarchies. The SOS compound can serve the same purpose as a directory, by aggregating content. Unlike the file system directory that can contain files and other directories only, the SOS compound can aggregate any SOS entity, as long as it is addressable within the SOS namespace. An important property of the SOS is that content having the same GUID is de-duplicated, irrespective of the compound it is referred from. De-duplication, in theory, could be implemented over standard file systems too, by using hard-links for every file that already exists on the file system more than once. However, de-duplication can be counterproductive in a file system, since two identical files contained in two different folders might have two different semantic meanings to a user, and therefore might be updated and used differently. The SOS can address this issue by abstracting compounds and atoms through assets and having compounds of assets, rather than compounds of atoms and other compounds (see Section 5.1.4.3[*Versions with Compounds of Versions*]). Managing compounds of assets, however, can be computationally expensive and inefficient.

7.3 Networked File Systems

Networked file systems can be categorised into: client-server file systems, clustered file systems, and shared-nothing file systems. This section compares each type of these file systems against the SOS in terms of abstractions and/or architecture.

7.3.1 Client-Server File Systems

Client-server file systems are designed to work over a limited number of servers and clients. The most common CSFS are the Network File System (NFS) [97], the Andrew File System (AFS) [98], and the Microsoft Server Message Block (SMB) [99, 100]. The SOS storage model will be compared against these file systems, considering three characteristics: the namespace, the service type, and the data access model (see Section 2.2.4.1).

Namespace

CSFS can have either global or per-client namespaces. AFS supports a global namespace, while NFS and SMB a per-client one. A per-client namespace is easier to manage and implement at the client-side, but requires an additional global namespace abstraction layer to avoid naming conflicts. The advantage of a global namespace is that it already provides location transparency. Names in a global namespace are generated by the server or automatically through the chosen naming scheme. The SOS also provides a global namespace, which originates from the way content is addressed.

Service Type

A CSFS can provide either a stateless or a stateful service. File systems providing stateful services, such as AFS and SMB, need to maintain the state of requests for a given session, which can be expensive and a bottleneck when scaling the system. Having a stateful service, however, can also reduce the number of requests made by the client. On the other hand, a stateless service, such as NFS, scales better since no state needs to be maintained across requests. The SOS provides a stateless service.

Data Access Model

CSFS support two types of data access models: the remote access model and the upload/download model. NFS and SMB provide a remote access model, while AFS the upload/download model. The SOS also supports the upload/download model, which is simpler and better suited for a large distributed storage system. A deficiency of the upload/download model is that it can be expensive when working with large files, since these have to be downloaded/uploaded from/to the server on each session. In a remote access model, on the other hand, the files never leave the server and constant communication between the client and the server allows operations to be performed. The issue of the upload/download model regarding large files is handled by the SOS model by chunking data into relatively small atoms and grouping them together using the compound. Thus only the necessary content is transferred over the network, which can also be performed in parallel.

7.3.2 Clustered File Systems

A clustered file system, such as Lustre [106], OCFS2 [107], and GFS2 [108], is a networked file system where multiple servers form a storage pool over a "shared-disk". The advantage of a CFS is that data written/read by the servers is consistent, however the shared-disk is a single point of failure in the system. In addition, coordinating the data access of multiple clients becomes expensive very quickly, as the number of clients increases. The SOS design is based on a "shared-nothing" model (see below) and avoids these issues. Data consistency in the SOS is achieved by implementing an immutable data storage, with no updates.

7.3.3 Shared-Nothing File Systems

A shared-nothing file system (SNFS) is a networked file system built over a pool of storage nodes with a "shared-nothing" model (*i.e.*, each node has access to its own storage disks). The Google File System (GFS) [111], Apache's Hadoop File System (HDFS) [110], and GlusterFS [137] are all example of SNFS. The SOS has an asymmetrical architecture (*i.e.*, some nodes are more important or have different roles than others), like the first two, while GlusterFS is a symmetrical SNFS. The advantage of an asymmetrical architecture is that nodes can be specialised in performing particular tasks. Asymmetry, however, can also be a disadvantage. For example, a node might be overloaded or become a point of attack that can crash the entire system. Balancing and distributing system and network resources is important to avoid such cases from happening.

The coordinator node, in the GFS and HDFS, is the key element that separates metadata from the actual storage of files and allows these systems to scale to thousands of nodes and billions of files. The SOS, in a similar fashion, decouples storage from metadata and node discovery management. The elastic hashing algorithm used by GlusterFS is an alternative approach to distributed content without the need of a metadata node. This approach, however, is not feasible for the SOS, where metadata is used to locate as well as describe data. It should also be noted that the GFS and HDFS rely on a single coordinator or multiple synchronised coordinators, while in the SOS nodes synchronise under the eventual consistency paradigm. The side effect of this is that clients accessing the SOS at the same time, may have different views of the content stored.

Another technique adopted by SNFS to manage large files is data chunking, which enables content to be downloaded/uploaded in parallel and easily moved and replicated across storage nodes. The SOS model also allows content to be chunked as well as deduplicated. In GFS, HDFS, and GlusterFS, data moves because requested by the clients or because more storage nodes are added or fail, leading the system to create new replicas of the lost files. In the SOS, similarly, data moves when requested by other nodes. Content replication and erasure coding can be achieved in the SOS through the context model.

7.4 Versioning in Storage Systems

7.4.1 Backup Applications

Backup applications, or services, are solutions that store and manage data so that users can recover previous versions of their data and/or recover lost content if the local storage fails. This section evaluates the versioning ability of such systems. Time Machine, as well as other solutions (*e.g.*, rsync[104], tarsnap[105], idrive[106], and syncthing[107]), provides the ability to set destination backup locations which are used to periodically store snapshots of the data. If the locations are not available (*e.g.*, the disk is not connected, the network connection is down, *etc.*), then the the backup operation is skipped.

7.4.1.1 Data and Aggregation Metaphors

The main advantage of backup applications over the SOS is that their data model is based on the same metaphors used by the file systems: files and directories. The SOS, instead, uses a different set of metaphors which users are not familiar with, such as the asset, compound, and the atom. However, as illustrated in the *Reference Implementation* Chapter, Section 6.4, the SOS model can be mapped to the more familiar file and directory metaphors, which are then presented to users and applications.

The next subsection evaluates how versioning is implemented in backup applications and how it compares to versioning in the SOS.

7.4.1.2 Versioning

One property supported by most backup applications is to apply versioning over the stored backups. Time Machine, for instance, stores backup snapshots, or versions, incrementally, using hard-links to de-duplicate files that have not changed.

The main difference between the versioning model adopted by backup applications and the one adopted by the SOS is that the former is based on the linear evolution of data, while the latter allows data to evolve in multiple ways thus supporting operations like branching and merging. Furthermore, the SOS files (via atoms) and directories (via compounds) can be versioned independently. The simpler versioning model used by backup applications, however, can also been seen as an advantage from the users' perspective, since it requires a gentle learning curve.

[104] rsync. `https://rsync.samba.org/` [last accessed on 20/03/2018].

[105] Tarsnap - Online backups for the truly paranoid. `https://tarsnap.com/` [last accessed on 22/04/2018].

[106] IDrive - Online Cloud Backups. `https://idrive.com/` [last accessed on 22/04/2018].

[107] Syncthing. `https://syncthing.net` [last accessed on 16/03/2018].

7.4.1.3 Data Backups via Replication

The SOS can provide the ability to backup data through the usage of contexts. For instance, a context with an *acceptAll* predicate (see Section 5.1.7.2) can have a replication policy that matches the behaviour of Time Machine, so that all data managed by a node is replicated (or backed-up) to the user's chosen destinations. In the SOS, data is replicated across nodes; while in backup applications, such as Time Machine, data is stored across storage disks, which are attached to a machine either locally or through the network. Time Machine enables users to select what data to backup or not. The same functionality can be achieved in the SOS by specifying *ad-hoc* rules within predicates. The SOS predicates provide a richer syntax than backup applications, plus the SOS policies enable users to backup different content in different ways.

7.4.2 Version Control Systems

The version metaphor used by the SOS is the same one used by VCS, such as git and mercurial, where each version stores references to its previous ones (see Section 2.2.3 about the different version metaphors). The main difference between VCS and the SOS, however, is not in the version metaphor adopted but on the level of granularity of versioning. In VCS, data is versioned in repositories, while in the SOS data is not linked to any repository or a similar type of containment. Thus, VCS operations like *pull* and *push* change their meaning when applied to the SOS model, since they should not be applied at the repository level but rather at the asset level. Thus, assets are pulled, pushed, branched, and merged, rather than entire repositories (see Section 5.1.4.3).

The architecture of the SOS is decentralised, similarly to decentralised VCS. In decentralised VCS, the involved actors own the full copy of a repository, while in the SOS a node might store only a partial set of all the modelled content. The advantage of this approach is that a node does not have to hold a copy of the entire SOS and therefore it is possible to manage larger quantities of data. On the other hand, no single node has a global, consistent, view of the SOS world and resolving conflicts can be hard. Moreover, SOS users are able to retrieve specific assets as long as the nodes storing them are reachable, while in decentralised VCS this problem is non-existent (assuming that a node has the latest version of a repository).

The SOS model is based on the Merkle DAG data structure like git and mercurial. The next two subsections compare the SOS against git and mercurial.

7.4.2.1 Git

Data Model

The git and SOS data models are very similar and share properties which are discussed below. Figure 7.1 compares graphically the two data models. First of all, both git and the SOS represent entities using persistent manifest-like data structures. In git, the manifests (*i.e.*, blobs, trees, commits, and tags) are stored using an *ad-hoc* format (see Section 3.3.3.3), while the SOS uses JSON, which is a more known and used format, but less compact in nature.

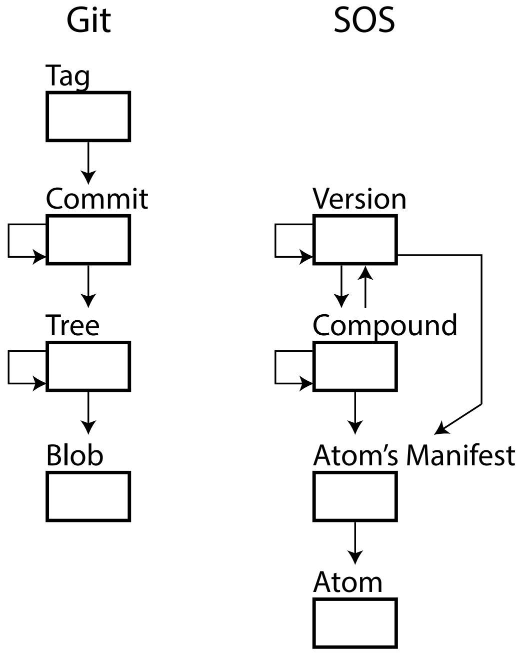


Figure 7.1: Comparison between the git and SOS data models.

Data

Git models data through the blob, which constitutes the building block of its model. Storing data in blobs, or blocks, is a common technique used in other storage systems too (see Chapter 3), which helps exploiting de-duplication, especially for large amounts of data. The approach of the SOS for modelling data is to use an additional indirection when addressing data (*i.e.*, atoms) via the atom manifest. The SOS has been designed to model data in a distributed environment, where users have only partial information about the entire system or even about a given asset. The atom manifest allows users to own a placeholder for the data, which also contains the information to retrieve the data (and decrypt it, if the atom is encrypted). While having an additional abstraction for data can be considered a disadvantage for some cases, this can be leveraged by using a distributed hash table over atoms directly.

Aggregation

Aggregation, in git and the SOS, is achieved via the tree and compound entities, respectively. The schemes adopted for the tree (git) and compound (SOS) entities differ slightly, but both of them collect content together by reference. Moreover, both designs allow content to be de-duplicated.

The main difference between the tree and the compound entities is that the former aggregates blobs and trees only, while the latter can be used to aggregate any other entity of the SOS. Moreover, the tree specifies the permission associated with each of its contents, while the compound simply allows content to be labelled.

Versioning

Git and the SOS provide versioning through the commit and asset entities, respectively. Both the commit and the asset allow content to be versioned over time, branched, and merged. The main difference between the two approaches is that the multiple versions of an asset are grouped together via the invariant GUID, while commits do not store any information regarding the repository they belong to, thus making it hard to distribute commits independently of their repository.

Metadata

In git metadata can be stored in two ways: via the commit or using notes. The SOS, instead, provides an explicit metadata model, where metadata is decoupled from the data it describes. Figure 7.2 compares graphically the metadata models for git and the SOS.

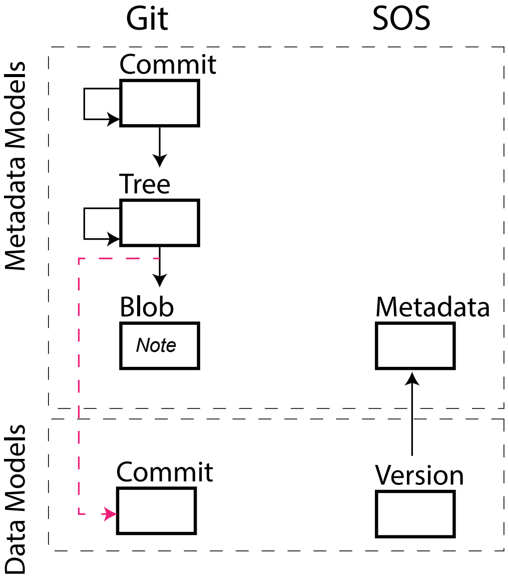


Figure 7.2: Comparison between the git and SOS metadata models and how they relate to the data models.

In git, the commit stores references (as email addresses usually) of its committer and author, the timestamp of its creation, and an arbitrary message specified by the user. Similarly, git notes can be used to associate additional information to a given commit. Except for the information about the committer, the author, and the commit's timestamp, git does not provide any well-defined structure about how metadata should be stored. The ability to write arbitrary commit messages and git notes results in a system where metadata can be inconsistent, scattered, and hard to understand.

In the SOS, metadata is modelled via the metadata manifest and its mutability is modelled through the asset, similarly to the asset's content. The SOS allows typed metadata properties (*i.e.*, string, int, boolean, *etc.*). One type of metadata property is GUID, such that it is possible to express metadata as reference values to other entities of the SOS, thus enabling richer relationships between entities of the SOS to exist. For example, the author or committer of a given version, in the SOS, could be expressed as a reference to a SOS role. The same approach is not possible in git, where authors and committers are simply text strings.

Internal data structures

The types of internal data structures used to store the git and SOS data models are strictly related to the actual implementations of the git tool and the SOS prototype. Git is much more mature than the SOS, thus it uses internal data structures that are more space efficient and optimised for IO operations. The git tool also provides a wider set of features than the SOS and it requires a large set of internal data structures which are not examined in this work.

Git stores the entities of its model in two ways: as objects and as packs. **Objects** are commits, trees, blobs, and tags of the repository (see Figure 3.9) that are stored in a compressed format. However, most of the git objects are smaller than the file system blocks, meaning that most of the file system block space is wasted. As a workaround, git uses **packs**, which are files collating together multiple objects. Packs are indexed to provide faster object reading.

In the SOS, manifests and atoms are not compressed and no packing mechanism is used, thus its space efficiency on disk is poorer than git. The lack of these optimisations can have a significant impact on disk usage since a large portion of the SOS entities are manifests, which are smaller than the file system blocks.

7.4.2.2 Mercurial

Data Model

This section compares the mercurial and SOS data models. The mercurial data model is different to the one of the SOS, since the data, aggregation, and versioning metaphors are not represented as manifest-like structures, but through the revlog data structure (see Section 3.3.3.4). Figure 7.3 compares graphically the two data models.

Mercurial consists of three abstractions: the *file*, the *manifest*, and the *changeset*. These abstractions will be written in italics, to avoid confusion with the more common file abstraction used in file systems and the manifest data structure used by the SOS.

Data

Mercurial models data through the *file* metaphor. Unlike the file metaphor used by

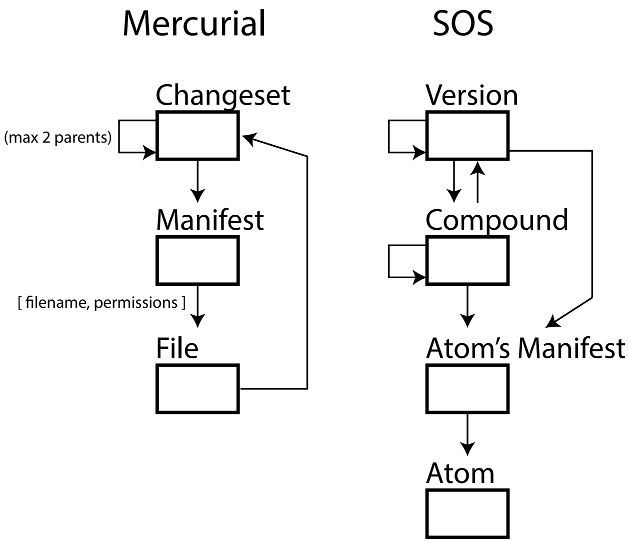


Figure 7.3: Comparison between the mercurial and SOS data models.

file systems, the mercurial one acts as a container for all the versions of a file. Its internal representation consists of a packed object containing multiple versions of the file as explained in Section 3.3.3.4. The collection of all the versions of a file forms a DAG, with each version referenced by a *manifest*. Furthermore, the mercurial *file* metaphor contains a reference to the *changeset* it belongs to, thus enabling a better navigation of the model. In the SOS no file abstraction exists, but one can be created through a combination of its data model components. A direct advantage of the SOS is that it abstracts the location of the atoms, such that data can be distributed across multiple nodes.

Aggregation

Mercurial models aggregation using the *manifest* entity. In the SOS, or git, the hierarchical structure of an asset, or a repository, is represented by nested compounds and trees. In mercurial, on the other hand, the hierarchical structure of the stored data is represented via the full pathname of a file, as stored in the *manifest*. In addition, the mercurial implementation stores files using a folder structure that reflects the pathname of the files. A disadvantage of the mercurial *manifest* is that it is not possible to exploit de-duplication at the *file* or *manifest* level. This can have an important impact when managing large non-textual data that is stored in multiple locations within the same mercurial repository.

Mercurial stores the information about *file* names and their permissions within the *manifest*, similarly to git's trees.

Versioning

The mercurial versioning metaphor, the *changeset*, is similar to the one adopted by the SOS, via the version manifest. Both metaphors describe a Merkle data structure, where the versioning entity originates from previous versions (if any) and its id is generated from these references plus the content of this version. The mercurial versioning approach, however, is based on storing the *deltas* of each version, which are applied one after the other as needed when a specific version is to be retrieved. The direct advantage of this mechanism is that no duplicate data is stored, but applying multiple deltas can be expensive. Moreover, no cross-file de-duplication exists. The SOS can store data efficiently by chunking it into fixed-size atoms and grouping them in a data type compound.

Metadata

Mercurial allows metadata to be modelled via the commit only (*i.e.*, a particular version of the changeset). Similarly to git, the commit can store only limited structured metadata, the timestamp and a reference to the committer, and unstructured information via the commit message. The SOS model, on the other hand, de-couples metadata from the content it describes and allows richer description of the content itself. The downside of this approach is that retrieving data and metadata requires multiple requests to the SOS.

7.5 Cloud Storage

7.5.1 Infrastructure as a Service Storage

This section provides a comparison between the SOS and IaaS object storage services, such as Amazon S3 [20], Backblaze B2 [125], and DigitalOcean Spaces [120].

7.5.1.1 Data and Metadata Model

The data and metadata models used by IaaS storage are more or less the same across different solutions and consist of objects made of raw data and structured metadata, which is usually a list of attribute-value pairs. Objects are stored in buckets that are not part of the model, but still play an important role in how objects are isolated and shared. Figure 7.4 compares graphically the IaaS model against the SOS data-metadata model.

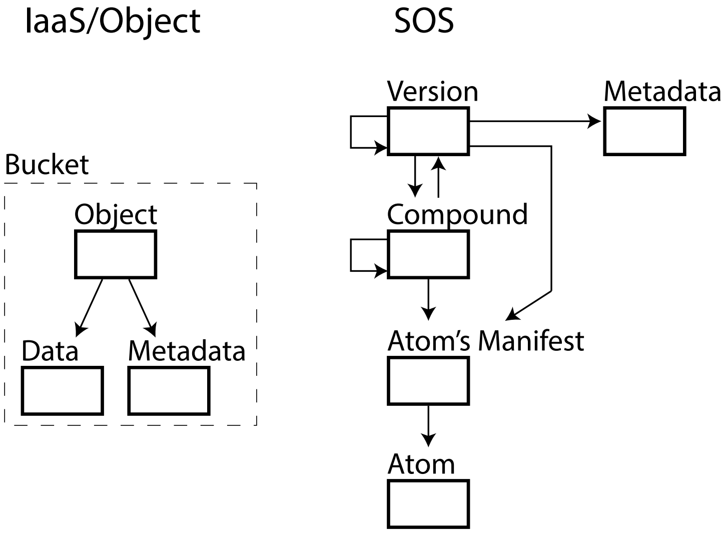


Figure 7.4: Comparison between the IaaS/Object storage and SOS data-metadata model.

IaaS storage provides a very simple data abstraction through the object, which encapsulates both the data and the associated metadata. The SOS, by comparison, provides a more complex data abstraction, via the atom and the atom manifest. In the SOS, however, atoms are not bound to specific nodes, while in IaaS storage, objects are bound to buckets. The consequence of the bucket-driven design of IaaS storage is that the bucket becomes the only source of truth, while in the SOS any node can store some given content. An advantage of the bucket design, however, is the ability to provide isolation for the data contained within a given bucket.

Most IaaS services also provide object versioning, modelled by assigning a version identifier to a particular version of an object. The most common IaaS storage solutions provide linear versioning only, while the SOS asset model is more flexible and allows operations such as branching and merging. The advantages of using a linear versioning model are simplicity and better ease-of-use for the IaaS' clients. The versioning model of the SOS is more complex, but also more expressive and allows its users to work on the same assets at the same time, without the need to constantly synchronise them.

Metadata in IaaS storage is bound to data, while in the SOS it is de-coupled from data and the asset. The advantage of the IaaS metadata model is that it is a simple design, easy to understand, and content and metadata can be moved across buckets together. However, services usually provide only limited functionalities over the metadata that can be stored for a given object. For example, it is not possible to define the type of metadata attributes and only a limited number of attributes can be set. A relevant difference between IaaS storage and the SOS is that the latter allows metadata to be distributed independently of the object it describes.

7.5.1.2 Access Control

IaaS storage solutions provide access control mechanisms similar to ACLs (access control lists). On Amazon AWS, for example, the IAM (Identity and Access Management) service is used to enforce access control across multiple AWS services. For Amazon S3, it is possible to control access both at the bucket and the object level. DigitalOcean's Spaces and Blackblaze B2 support a similar access control mechanism to Amazon S3.

The abstractions used by the SOS to enforce access control are the user and the role. While these abstractions are similar, AWS is able to provide a wider set of functionalities and enforce different levels of access for users. For example, on AWS it is possible to group users together and define what operations a user is allowed to perform on a given service, bucket, or object. Grouping together users or roles is not supported by the SOS. The ability to associate specific operations with users/roles is also not supported.

7.5.1.3 Data Policies

The ability to define data policies is an important feature provided by some IaaS object storage solutions because it allows users to establish how content should be stored, replicated, and protected over time. The SOS can provide a similar abstraction via contexts. Amazon S3 policies provide two main functions: data migration and data replication to other Amazon S3 buckets or Amazon storage services (*i.e.*, Amazon S3 Standard IA or Amazon Glacier). Similarly, Google Cloud Storage uses policies to manage the life cycle of the stored content.

The SOS does not provide migration and replication policies by default, but it allows users to specify such policies within the context model, such that the same behaviour is obtained. The SOS policy model is more flexible and allows users to specify richer and more diverse policies. On the other hand, IaaS are generally trusted services and their policies are well-tested before being pushed into production. The same level of trust over policies cannot be enforced in the SOS.

Furthermore, services like Amazon S3, Azure Storage, and Google Storage provide a notification service, so that other services are notified when an event occurs on the storage layer (*i.e.*, when a stored object changes).

7.5.1.4 Architecture

The general architecture of IaaS storage services is principally based on the distribution of storage machines across multiple locations around the globe. Locations are typically grouped into regions, each consisting of multiple data centres. Moreover, some services have started adding *edge* locations to bring storage closer to users.

The exact inner details of the commercial storage services are not known. IaaS storage services, however, can be built on top of shared-nothing file systems like GFS, HDFS, GlusterFS, or Ceph.

The SOS architecture is based on a fully decentralised network of nodes. IaaS storage services can also implement a distributed architecture for better availability, redundancy, and resiliency. However, a decentralised IaaS architecture is still subject to the vendor lock-in issue. The SOS attempts to solve the vendor lock-in problem by providing generic abstractions, which can be implemented by any storage provider. On the other hand, enforcing control over data is easier in an IaaS architecture, since all its components are known and under the same control.

7.5.2 Software as a Service Storage

This section provides a comparison between the SOS and SaaS storage providers, such as Dropbox [18], Google Drive [124], and OneDrive [19]

7.5.2.1 Data Model

SaaS storage providers, such as Dropbox, Google Drive, and OneDrive, use the file and directory metaphors when presenting content to users. These metaphors are well-known and easy to use, but they no longer represent the workflows and needs of today, as outlined in [14].

Often SaaS storage providers are also associated with additional abstractions provided by application-centric systems, such as Android, iOS, or the Windows 8's metro. Application-centric abstractions facilitate data-usage, especially within the specific scope of an application, by completely hiding their internal representations. However, application-centric systems are still based on the common file/directory or object metaphors, so the limitations of the latter still persist.

Cloud storage solutions, such as Dropbox, Google, and Adobe, have also designed new abstractions that allow users to work with file's placeholders, so that data does not have to be necessarily stored by the user, unless this must be accessed. The Adobe DCX format [168] allows content to be bundled together using established semantics. The SOS compound abstraction is comparable to DCX, but its semantics is simpler. Additionally, Goldman [168] suggests that data modelled by DCX can be controlled using *ad-hoc* algorithms to provide partial data synchronisation, parallel data uploads/downloads, or simply to enable storage context-awareness. The Microsoft file biography abstraction [167] is an attempt to extend the file in terms of the functionalities it provides in a distributed environment, but it fails to describe how data is modelled and how the suggested interactions should be implemented.

The SOS provides metaphors that are partially inspired by the ones provided by VCS, which themselves are less known to most users. However, applications and services built on top of the SOS can abstract these metaphors and present content to users via the more classic file and directory metaphors, as shown in Section 6.4.

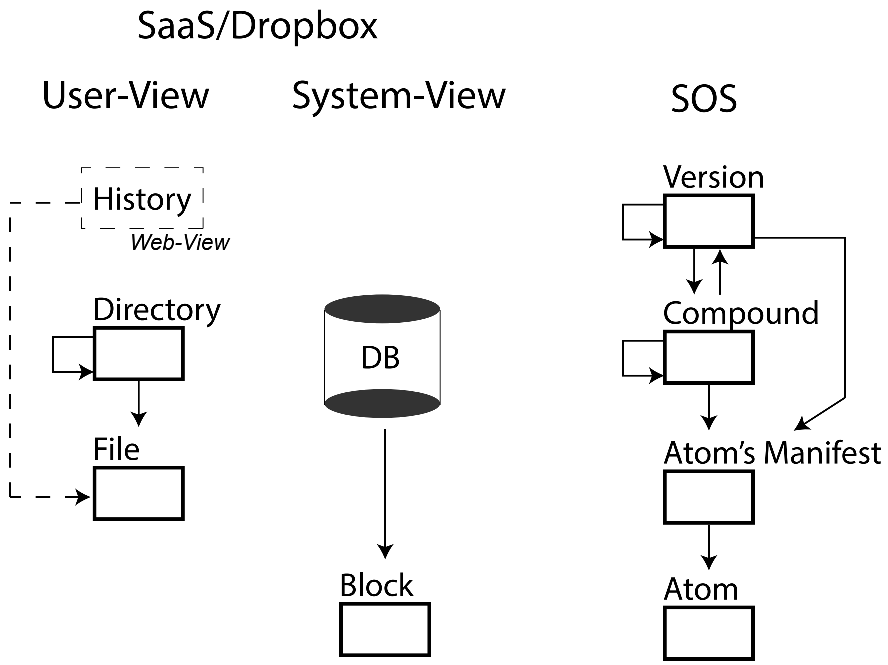


Figure 7.5: Comparison between the Dropbox and SOS data models.

Versioning

Most SaaS storage solutions also provide content versioning, which is presented to users via the browser or *ad-hoc* applications due to the limitation of the current file system abstraction. The underlying versioning models used by the most relevant SaaS solutions are unknown and therefore they cannot be compared to the SOS versioning approach.

The Microsoft file biography proposes a versioning abstraction based on multiple snapshots of a file, which can be marked as edits, milestones, or actions such as the file being sent over to someone, edits merged, or the file being deleted (or withdrawn) from a given location. Some of these operations are directly comparable to the ones provided by VCS or the SOS, such as committing, pushing, pulling, merging, branching. Deletion or withdrawn operations, instead, are not comparable. Unfortunately, the authors [167] present the file biography metaphor from an end-user perspective only, without providing a description of a supporting model.

Metadata

SaaS storage solutions usually store basic metadata regarding the names of the stored files and their locations on different client machines. Some solutions allow users to specify arbitrary metadata attributes, which can then be inspected through their web applications. The file biography [167] and the DCX format [168] are two efforts by Microsoft and Adobe respectively to extend the file metaphor, not only in terms of the functionalities they provide, but also in terms of the metadata it can record. The SOS, however, takes an alternative approach to model metadata, where this is de-coupled from the content it describes. Similarly, the history of some data, which is modelled through the asset entity, is also a type of metadata that is de-coupled from content. The advantage of de-coupling metadata from content is that one may distribute one irrespectively of the other. This allows the SOS to have distinct specialised nodes for managing metadata and data.

7.5.2.2 Architecture

The architecture of most relevant SaaS storage solutions is unknown, except for the Dropbox one which has been partially presented through the official blog of the company and studied over the years by researchers in the academia.

Dropbox stores data on data servers that are clearly separated from the ones managing the metadata. This approach is similar to the GFS or the HDFS, where metadata and data are stored and managed by two different types of nodes too. Dropbox avoids storing the same content multiple times by de-duplicating content, but it applies this technique on a per-user scope only, to avoid global de-duplication attacks [158]. The current design of the SOS provides content de-duplication too, but it does not prevent global de-duplication attacks.

The underlying storage system used by Dropbox — Magic Pocket — is designed specifically to exploit de-duplication and improve IO performance when data is moved/copied between replica machines. The reference implementation of the SOS presented in this work does not use any particular optimisation technique to efficiently store data on disk.

Finally, Dropbox client's LAN Sync feature can be used to exchange data directly between nodes residing in the same network. The SOS is based on a P2P architecture, thus it can easily provide a similar functionality since nodes are distinguished for the services and content they provide, rather than on the organisation they belong to.

7.5.3 Multi-Cloud Storage

The goal of multi-cloud storage is to abstract IaaS and SaaS storage solutions, so that end-users should not be concerned about the services involved and where data is located. The SOS provides data location abstraction that allows data to exist on any type of node, as long as it belongs to the SOS network. SOS nodes can also be easily integrated with existing storage services, as discussed in the *Reference Implementation* Chapter. Moreover, while multi-cloud storage solutions strive to design a data model that fits existing cloud storage services, the SOS introduces a new data model to which storage services should adhere to. This can be a disadvantage, since services and applications need to map their solutions to the SOS model, rather than the opposite.

7.6 P2P

In this section, the SOS is compared against the P2P storage system presented in Section 3.5.2.

7.6.1 OceanStore and Pond

7.6.1.1 Data and Metadata Model

The SOS and OceanStore [195, 112] data models are two different implementations of a Merkle DAG. Both models provide versioning, data location abstraction, and content aggregation.

Both are content-addressable systems and allow content to be aggregated together. OceanStore and the SOS support mutability by aggregating versioned content via a common, invariant, GUID (AGUID in OceanStore). The versioning property of the OceanStore model, however, is stricter than the SOS one, since data is versioned only linearly, while the SOS data model supports a more flexible model, where branching and merging operations are possible. The linear versioning model used by OceanStore is supported by the use of a Byzantine agreement protocol implemented by the primary replica peers. In the SOS, instead, assets can be updated concurrently by multiple users at the cost of sacrificing Byzantine fault tolerance.

Both OceanStore and the SOS allow users to associate metadata with each version of some content. In OceanStore metadata is embedded within the version object, while in the SOS the metadata is de-coupled from the version. Embedding the metadata with the version object results in a simpler model and the ability to easily distribute versions and metadata together. The SOS approach, however, favours the distribution of metadata independently of the data it describes. It is also possible to de-duplicate metadata across multiple versions and/or assets, which is particularly useful when having large metadata.

7.6.1.2 Access Control

OceanStore enforces access control via the primary replica peers, which either distribute the data to other nodes (read permission) or maintain an ACL (write permission). In OceanStore, users and nodes are one and the same. In the SOS, on the other hand, users are modelled independently of the nodes they use. Thus, users are able to interact with the SOS independently of the node they use. The disadvantage of this approach is that the SOS has to store and manage an additional model. This results in extra operations when writing/reading protected (or signed) content. A separate user-role model, however, abstracts protections and ownership from node locations and enables better control over the content to be protected, in terms of granularity, ability to use different encryption algorithms, and ease by which permissions are granted. The SOS, however, does not implement access control mechanisms to differentiate between read and write permissions.

7.6.2 InterPlanetary File System

7.6.2.1 IPLD and the Multiformat Project

The basic building-block of IPFS [113] is its generic data model framework, IPLD [198], which describes how entities are represented, and the multihash format [223], which defines how content is named and addressed. Similarly, the SOS is based on the manifest data structure and the SOS GUID format.

Both IPLD and the SOS manifest structure allow content to be recorded and referenced. IPLD's Merkle-paths are equivalent to SOS paths. The current advantage of IPLD is that data structures can be expressed in multiple formats (*e.g.*, JSON, XML, YML), while the SOS supports JSON manifests only. This encourages the adoption of the model across multiple, and possibly existing, applications and services.

The multihash format used by IPLD and IPFS is very similar to the SOS GUID. Both formats rely on their users knowing the association between hash functions and hash codes, for the multihash, or hash string names, for the SOS. The multihash has been designed to be compact, while the SOS GUID occupies more space, but it is easier to read and parse, since it also describes the hash value base. The multihash format belongs to the multiformat protocol and while it is not an established standard yet, it has recently gained relevant traction from the open source community.

	Multihash	**SOS GUID**
Algorithm	Unsigned integer (1 byte)	String (variable length)
Length	Integer	n/a
Base	n/a	Integer (*e.g.*, 16, 32, 64)
Value	Expressed using the base that matches the hash length	Expressed using the specified base

Table 7.1: Comparison between Multihash and the SOS GUID formats.

7.6.2.2 Data Model

The IPFS data model, which is based on the git model, consists of four entities: the block, the list, the tree, and the commit. The IPFS entities are directly comparable to the SOS ones (see Figure 7.6). Both models are able to abstract data from locations, either via the blob or the atom. The SOS atom manifest, however, can be used as a placeholder metaphor at the cost of adding an additional level of indirection when retrieving the data. The IPFS model does not enforce any size limits on the blob, but the IPFS tool shards data into small (256KB maximum) blobs by default, which are grouped in a list, to exploit de-duplication. The SOS prototype does not provide any sharding functionality, but it is theoretically possible to achieve the same result by using multiple atoms and grouping them using a *data type compound*.

Both IPFS and the SOS allow aggregation by reference, via the list and tree for the former and the compound for the latter. While the SOS compound stores content always in order, by GUID, the IPFS provides two aggregation entities, the tree and the list, for unordered and ordered content. In the original IPFS design, trees and lists could contain blobs and other trees and lists, but not commits. This is a restriction of IPFS only, since with the introduction of IPLD each entity can contain references to any other entity of the model.

IPFS models versioning using commits, similarly to git. The SOS asset provides the same versioning capabilities of the IPFS commit, but in addition it stores a GUID unique to all its versions, the invariant, and it allows content to be associated with metadata.

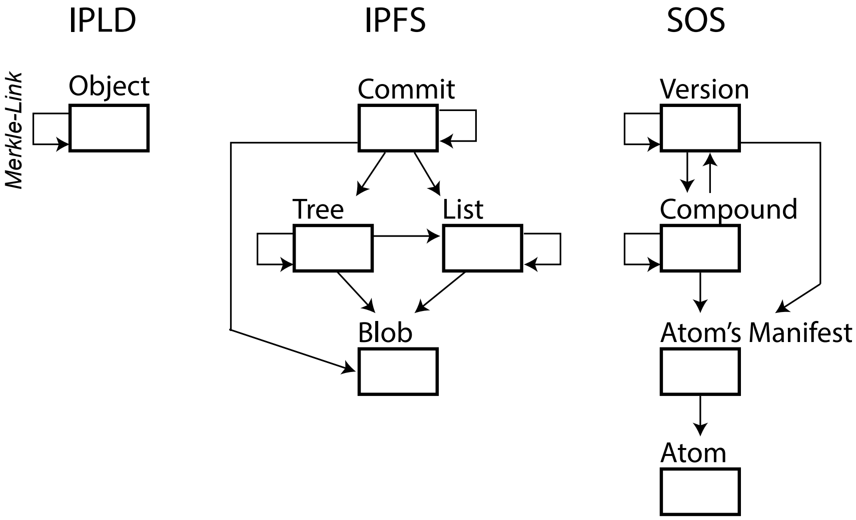


Figure 7.6: Comparison between the IPLD/IPFS and SOS data models.

Both IPFS and the SOS sign and protect content in-transit using the keys of a node. The SOS, however, allows content to be signed and protected at-rest too, using the role abstraction, which exists and can operate independently of a given node.

7.6.3 Perkeep

7.6.3.1 Data Model

The Perkeep [190] data model is based on content-addressable entities, but the types of entities, their relationships, and how they are used to define aggregation and versioning are very different from the SOS or similar systems, such as IPFS and OceanStore.

Aggregation

Perkeep achieves aggregation using three types of entities: directory-claims, files, and static-sets. The static-set is equivalent to the SOS compound of *collection* type, while the file entity is comparable to a SOS compound of *data* type. The directory-claim is a third type of aggregation, where it is its content that references the directory (*i.e.*, the permanode associated to the directory-claim), rather than the opposite. The directory-claim is comparable to a versioned compound that aggregates assets (see section 5.1.4.3[*Versions with Compounds of Versions*]). In the SOS approach, when a compound is updated, the change has to be propagated to any compounds containing it, an operation that can be computationally expensive. While in Perkeep, when the contents of a directory are updated, the former will simply reference the permanode of the latter, which does not change.

Versioning

Perkeep models versioning via claims and permanodes. The permanode acts as the immutable reference for all the versions, similarly to the invariant GUID for the SOS asset. However, while SOS versions are linked together forming a Merkle DAG, the Perkeep claims are related to each other only by referencing the same permanode and are simply ordered via timestamps. The Perkeep approach results in linear-versioning only. In addition, a distributed system cannot safely enforce ordering only using timestamps, without using any clock synchronisation mechanisms or logical clocks.

7.6.3.2 Access Control and Protection

The current Perkeep model does not support any explicit access control mechanism. Moreover, Perkeep supports data protection in-transit only.

Both Perkeep and the SOS define a user/role model to sign their entities. The main difference between the two models is that Perkeep supports unnamed users only, while the SOS defines named users, which are related to one or multiple roles. It is the role entity that then is used to sign as well as to protect content.

7.6.4 Tahoe-LAFS

7.6.4.1 Data Model

Tahoe-LAFS [191] and the SOS both support content-addressable content, which can be distributed over a network of nodes. In Tahoe-LAFS data is modelled around the standard file and directory metaphors, while the SOS extends the standard model to support versioning too. Moreover, the Tahoe-LAFS data model is designed to provide high levels of privacy and security, achieved by hashing content differently based on the access control to grant. Moreover, encryption, file chunking, and erasure coding are used to improve the level of data dispersion and resiliency. The use of a convergence secret, when generating the capabilities of files, is used as a protection mechanism against global de-duplication attacks.

The SOS employs content protection and access control through the user-role model, as described in Section 5.1.6, but it is not able to distinguish read-only and read-write access control like Tahoe-LAFS. Access to SOS-protected entities can be granted to multiple roles while preserving the same GUID, unlike Tahoe-LAFS where the id — or capability — of each file must be generated for every user. The disadvantage of the SOS approach is that a successful breach on a shared protected entity compromises that entity for everyone else who has access to it. Both systems do not provide a clear approach for revoking privileges on protected content.

Both Tahoe-LAFS and the SOS protect content at-rest and in-transit. Tahoe-LAFS encrypts, chunks, and applies erasure coding by default, while in the SOS encryption is optional as well as chunking and erasure coding. The reference implementation presented in this thesis, however, could be upgraded to provide such features by default.

7.7 Context-Aware Storage

The objective of context-aware storage is to provide users richer ways of managing and querying content. Over the last three decades many solutions have been proposed, from the Semantic File System [206] and WinFS [76], to the more recent quFile [209], Adobe DCX [168], and Microsoft file biography [167]. The SOS also enables context-aware storage via the context model.

The design of context-aware storage is based on the description of computation that allows content to be captured under some given semantics. Some systems also provide computation in terms of what to do with the captured content, such as caching, replicating content, extracting metadata, *etc.* These traits are also present in the SOS design, with the predicate and policy computational work units. In the SOS, computation is decoupled from the data model, while in some other systems, such as WinFS, GAIA [208], or Lifestreams [207], this separation is not very clear. De-coupling computation from data results in a system where computation is clear and can be distributed more easily. Systems like WinFS and GAIA, however, provide richer context-aware storage properties than the SOS. In GAIA, storage is built within a context-aware operating system, so that what data is presented to users depends on contextual factors such as user, time, light, temperature, number of people in a room, *etc.* In WinFS, the system is able to define strongly typed relationships. In the SOS it is also possible to establish relationships between assets, but the type of metadata is arbitrary. In Lifestreams, files are located along a time-axis, such that users are able to navigate the file system along such axis. In the SOS it is possible to achieve a similar behaviour, but the time location of assets would depend on when predicates are run.

Finally, an important aspect of the SOS context model is that it has been designed to work in a distributed environment, where predicates and policies are run over domains and codomains of nodes. Amazon S3 also allow users to define over which locations - at the bucket level - policies can act upon.

The next two subsections compare in more detail the SOS with the Semantic File System and the quFile. The next chapter discusses the SOS context model in more detail as it has been evaluated experimentally.

7.7.1 The Semantic File System

7.7.1.1 Data Model

The SFS [206] data model is an extension of the classical file system one. Data is presented to users via the file metaphor, while content aggregation is achieved using virtual directories. Virtual directories, unlike normal directories, provide dynamic aggregation of content. In the SOS, aggregation can be achieved via immutable compounds or via contexts. The latter can be used to achieve the same behaviour of the SFS in terms of automatically aggregating content.

7.7.1.2 Transducers and Contexts

The SFS and the SOS provide computation over content via transducers and contexts, respectively. Both systems have been designed to automatically classify content, but present the following differences:

- Transducers can be run over files only, while contexts are run over assets, which are richer abstractions that are version-aware and can also refer to other entities (*e.g.*, atoms, metadata, roles).
- Transducers can only identify and aggregate files together, while the SOS can use policies to allow content to flow across nodes, thus achieving operations such as data migration or replication.
- The content identified by transducers is presented to users via virtual directories, which can be combined under the *AND* boolean operator. SOS contexts cannot be combined.

- Transducers are event-triggered, while the contexts are run on a periodic basis. An event-triggered system is conceptually simpler and easier to run, but in practice it requires platform dependent code to be able to capture relevant events from the operating system.
- Contexts are designed to be run over multiple nodes, having a source domain and a sink codomain of nodes. Transducers can be run on the local file system or any mounted network storage.

7.7.2 The quFile

The quFile [209] is a file metaphor designed primarily to aggregate multiple representations of a file and present users with one of them based on the context. The quFile metaphor is very different to the SOS asset, which is a collection of immutable versions of some data. The SOS can also support the aggregation of multiple representation of data via the compound, but is not able to differentiate one representation from another, unless using *ad-hoc* metadata. Moreover, each quFile is associated with a set of policies, which can be used to make better decisions regarding what data to return or what permissions users have on content. In the SOS, policies are defined for all the contents of a context, rather than on a per-asset level.

Figure 7.7 compares graphically the quFile and SOS data models as well as their computational models. First of all, the data model elements of the two systems are similar, in terms of how data is aggregated, except that the SOS can achieve content and multiple-representation aggregation using the compound only, while quFile storage uses the directory as well as the quFile metaphors. Moreover, the SOS allows content to be versioned and abstracted from locations. The context model of the quFile is simpler than the one proposed by the SOS. Understanding the former is simpler for end-users, since there are less components involved and policies are directly related to quFiles. The SOS context model, instead, is more modular, which can be an advantage in a distributed system where content is replicated over multiple nodes. In a very large system, for example, having de-duplicated predicates and policies can result in space and computing time-savings.

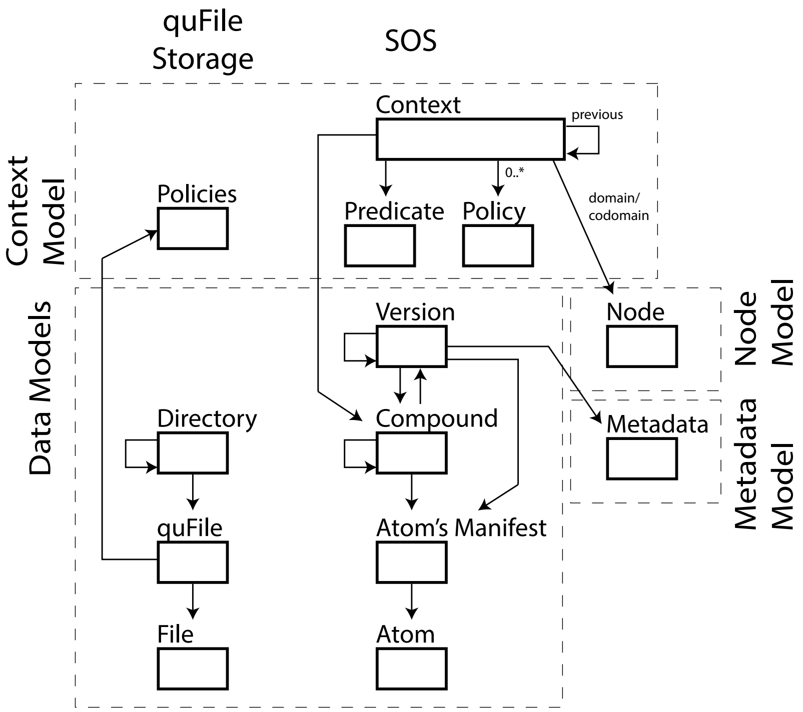


Figure 7.7: Comparison between the quFile and SOS abstractions.

7.8 Conclusions

This chapter has evaluated comparatively the design and architecture of the *Sea of Stuff* against the relevant data management systems (DMS) presented in the *Background* and *Literature Review* Chapters. First of all, the SOS model and implementation were evaluated against the requirements specified in Chapter 4. In the sections following, a qualitative comparison provided insights on the similarities and differences between the SOS and other systems. In particular, it has been shown that the SOS uses the same or similar data management techniques applied by other systems, which have proven themselves to be scalable and reliable over the years.

To summarise, the following remarks can be made regarding the SOS in relation to the relevant related work:

- The file and directory are the two most well-known storage abstractions, but it has become hard to understand what they are and how they behave when used in a distributed heterogeneous system. Many systems are trying to extend or re-think these metaphors (*e.g.*, file placeholder, Lifestreams, Microsoft file biography, *etc.*). This thesis proposes a well-defined model: the SOS. The SOS model consists of a variety of abstractions which allow data to be accessible irrespective of its locations, aggregated, versioned, digitally signed, protected, and automatically managed across nodes of the system. The SOS abstractions are analysed extensively in this chapter, presenting advantages and disadvantages of its design.

- The SOS allows content to be accessible irrespective of its locations and the location of users. Networked, cloud, and P2P storage can all provide such properties, but often still rely on the standard file metaphor, which does not describe location abstraction. The entities proposed by the SOS attempt to provide a metaphor that is alternative to the file and accessible irrespective of its locations.

- Versioning has become a fundamental property of many modern DMS. The solution provided by the SOS is similar to the one used by git, OceanStore, and IPFS. The asset entity also allows data to be versioned independently of where content is stored and other assets. In addition, the asset entity enables metadata to be linked to the content it describes.

- Data access control can be hard to manage over distributed data. The SOS attempts to control data access by modelling users as part of its model. Relevant information needed to sign or protect content is bound to a user, rather than stored within an ACL or a database. The current SOS user-role model has also limitations. For example, the SOS is unable to distinguish between read-only and read-write capabilities for data.

- The context model allows the SOS to provide context-aware storage. No standard model for context-aware storage has been established yet. The SOS attempts to provide a generic model which is technology agnostic and allows users to create rules to classify and automatically manage content in a distributed setting. Unlike other context-aware DMS, such as the Semantic File System, the context model does not have a well-defined and complete syntax for its rule-based components. Future work should focus on improving these aspects of the model.

- The architecture design of the SOS has been driven by the different components of its model. In the SOS there are six types of nodes (NMS, MDMS, URMS, MMS, SMS, CMS), as presented in Chapter 5. This reduces the presence of bottlenecks in the system and encourages the SOS to have nodes that are specialised on particular aspects of its model. This design was inspired by the coordinator-datastore architecture used by the GFS and HDFS, where the datastore simply stores and replicates the data, while the coordinator handles the metadata and periodically checks that the datastore nodes are functioning properly. However, with such architecture the system might become unreliable and unresponsive in the case of extensive multiple network partitions.

- The current SOS design, unlike the related DMS, is agnostic regarding the strategy to use about where to store/retrieve data. Extensive research has been done on P2P systems over the years. Future work on the SOS should explore the integration of P2P research with its model and provide further evaluation against the relevant DMS, such as OceanStore, IPFS, and Tahoe-LAFS.

A summary of the comparison presented in this chapter can be found in Appendix D in tabular form.

Experimental Evaluation

This chapter evaluates the *Sea of Stuff* reference implementation presented in Chapter 6. In particular, the experiments are designed around the research hypothesis **H2** presented in the *Introduction* Chapter. This chapter is organised as following: Section 8.1 lists the experiments presented in the remainder of the chapter; Section 8.2 outlines the methodology used to define the experiments; Section 8.3 describes the *testbed* used for the experiments; and Section 8.4 outlines the architecture of the experimental framework used to run the experiments over multiple machines. The experiments and their results are presented in Sections 8.5 and 8.6. Finally, Section 8.7 concludes the chapter.

8.1 Experiments Outline

This section outlines the experiments presented in this chapter. Two classes of experiments have been run: (1) basic benchmarks experiments on the prototype and (2) experiments aimed at understanding the performance and behaviour of contexts in a small distributed environment. The following list summarises the experiments:

Reference Implementation Benchmarks

- **Input/Output Benchmark** – Section 8.5.1. The following benchmark tests the reference implementation in terms of how well it performs when writing/reading atoms and manifests within the local node. The results of this experiment define the baseline performance of the reference implementation when data is read/written to the local storage.

- **Network Latency and Throughput** – Section 8.5.2. The latency and throughput between any two nodes running on the experimental *testbed* are measured and evaluated. The purpose of this experiment is to define the maximal throughput between any two nodes of the *testbed*, baseline needed to understand the results of other experiments that involve multiple nodes interacting with one another.
- **Atoms and Manifests Replication** – Section 8.5.3. This benchmark evaluates the performance of the SOS reference implementation when replicating atoms and manifests across multiple nodes. The results of this benchmark define the baseline performance for content replication, which is a primary element for the context experiments.

Context Experiments

- **The Nature of the Predicates** – Section 8.6.1. These experiments explore the relationship between the nature of the predicates and the time taken to identify contents for a given context. These experiments show that the feasibility of using contexts in the real-world depends on what predicates are being run.
- **Predicates and the Domain** – Section 8.6.2. These experiments demonstrate that the number of nodes in the domain and the amount of data to process by the predicate within the domain have an effect on the overall running time of the predicate. The relationship between these factors is studied in these experiments.
- **The Nature of the Policies** – Section 8.6.3. These experiments study the relationship between the nature of policies and the time taken to enforce them. These experiments show that the feasibility of using contexts in the real-world depends on what policies are being run.
- **Policies and the Codomain** – Section 8.6.4. These experiments investigate the relationship between the number of nodes in the codomain, the amount of data to control through the policies, and the overall running time to enforce them.

- **Contexts under Failure** – Section 8.6.5. These experiments demonstrate the efficacy of contexts in recovering from node failures.

8.2 Experimental Methodology

The experimental methodology used to design, define, run, and evaluate each experiment consists of the following steps:

1. The experiment is designed and described in terms of the following variables:
   (a) **Independent variables**: these are the variables that are changed during the experiment and will affect the dependent variables.
   (b) **Dependent variables**: these are the variables that are being observed and measured during the experiment.
   (c) **Controlled variables**: these are the variables that are kept constant and unchanged throughout the experiment, so that we can study only the effects of independent variables being tested.
2. The experimental setup is adjusted for the specific experiment in terms of the number of iterations to run, the nodes involved, and the dataset used.
3. The code defining the experiment is written in Java as described in Section 8.4.2.2.
4. The SOS node application, configuration files, and dataset are distributed and managed through the experimental framework, as described in Section 8.4.
5. The experimental data from the nodes is collected and analysed using R[108] as a statistical tool. A confidence interval of 0.95 has been used when analysing the data.

8.3 Experimental Setup

The experiments presented in this chapter have been run on the Butts Wynd Cluster (BWC), located at the University of St Andrews, UK. The BWC consists of 24 microservers, divided between two Dell™ C5220 PowerEdge Rack Servers. The two racks, named

[108] R - The R Project for Statistical Computing. `https://r-project.org/` [last accessed on 30/04/2018].

sif and hogun, have servers with dual quad-core Intel Xeon 3.4GHz CPUs, 16GB RAM, two 500GB SATA disks, and two Gigabit Ethernet interfaces. Each server, or node, is named using the following convention: ⟨sif/hogun⟩-⟨1-12⟩.

Each node has three main **storage** locations: cluster-home, cs-home and scratch-space. The cluster-home storage is accessible under /cs/⟨sif/hogun⟩/username/ and it has a capacity of 350GB. The cluster-home storage is shared across the nodes of the same rack. The cs-home storage is accessible under /cs/home/username/ and is a shared storage space between all nodes for the same user. Finally, the scratch-space storage is available under /cs/scratch/username/, has a capacity of 500GB and it is local to each node (*i.e.*, it is not shared across nodes). The experiments run for this work use the unshared space.

The head nodes of the cluster, sif-1 and hogun-1, are configured so that their resources are dedicated to manage the rest of the nodes in the rack. Due to the resources limitations of the head nodes, only the following sets of nodes have been used for the experiments: sif-⟨2-12⟩ and hogun-⟨2-12⟩.

The following is the relevant **software** specification used for each machine:

- Operating System: Scientific Linux 7.4 (Nitrogen), Kernel v.3.10.0
- Java SE Runtime Environment (JRE) 1.8 (build 1.8.0_161-b12). The following parameters were used for the JVM:
  - Initial heap size: 8GB
  - Maximum heap size: 12GB
  - Used the garbage collector G1 [224], which is designed to perform better when using multiprocessor machines with large amounts of memory

The BWC is a shared resource for staff and research students of the School of Computer Science at the University of St Andrews. No user management policy is enforced on the BWC. CPU, memory, network, and other users' usage were monitored using *collectd*[109],

[109] Collectd - The system statistics collection daemon. https://collectd.org/ [last access on 13/05/2018].

which was already installed and configured in the cluster. Additional information about the experimental setup can be found in Appendix B.

8.4 The Experimental Framework

8.4.1 Extension of the Sea of Stuff Node

The SOS node implementation, described in Chapter 6, contains a set of features for recording the performance of particular functions and controlling the behaviour of a node:

- The node has been instrumented, as described in Section 6.1.9.
- The context management service allows for predicates and policies to be triggered manually rather than through the scheduled thread pool.
- The REST interface has been extended so that some of the node's APIs can be disabled to simulate node failure. The REST APIs of a node can be re-enabled in a similar fashion.

8.4.2 The Experimental Framework Utility

The experimental framework utility (EFU) is a tool developed to describe and manage the experiments in a distributed setting, such as the BWC. The EFU consists of three elements: the experiment itself, which is described in Java, the distribution of the applications and datasets over the *testbed*, and the management of the experiment from a remote node, from start to data collection.

8.4.2.1 Experiment Distribution and Remote Management

When distributing and running an experiment, three types of nodes are involved: the setup node, the coordinator node, and the slave node. The **setup node** is responsible for (1) distributing the node application, the node configuration files, the experiment configuration file, and the datasets; and (2) starting/stopping the nodes, removing files from the nodes, and collecting the results of the experiments. The **coordinator** is the node that runs the experiment (see Section 8.4.2.2) and records the results through the node instrumentation. The **slave** is a node of the SOS network that takes part in the experiment, but does not record any measurements. Failure of a slave node can be triggered by the coordinator node. There can be zero, one, or more slave nodes for a given experiment.

The diagram in Figure 8.1 illustrates the interactions between the three types of nodes involved in running an experiment. The diagram shows how the SOS node application is distributed across the nodes in the cluster. The coordinator node, however, runs a different application which contains the logic of the actual experiment (see Section 8.4.2.2). The dataset is also transferred from the setup node to the coordinator and slave nodes. Finally, the coordinator checks that the slave nodes are up and running before each iteration of the experiment is started.

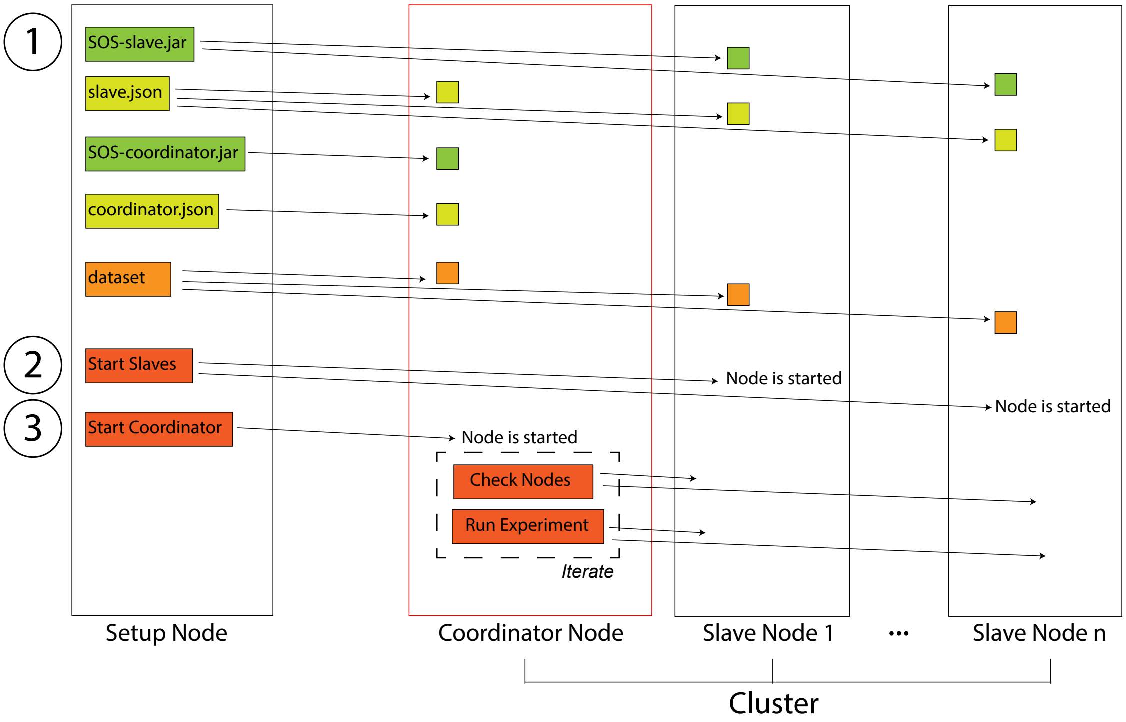


Figure 8.1: Diagram showing how the experiment framework (1) distributes the SOS node application and the necessary configuration over the nodes used for the experiment; (2) starts the nodes; and (3) starts the experiment on the experiment node.

8.4.2.2 The Experimental Structure

Experiments are defined in Java and are executed in three phases:

- **Setup**, where the experiment configuration file is processed and the instrumentation enabled. The experiment configuration contains information regarding the datasets available for the experiment, what code to instrument, and the nodes involved in the experiment.
- **Running**, when the experiment is run. The experiment is run over multiple **iterations**. The coordinator node is restarted at each iteration.
- **Finish**, where the resources, used by the experiment, are cleaned up.

The datasets used for the experiments are described in the next section.

8.4.3 The Datasets

The experiments presented in this chapter are run against the following datasets:

- **Random datasets**. The contents of this dataset were randomly generated using the *synthetic dataset tool*.[110] Multiple random datasets were used:
  - **R_100KBx100** – This dataset consists of 100 files of 100KB in size each.
  - **R_1MBx1** – This dataset consists of a single file of 1MB in size.
  - **R_1MBx1000** – This dataset consists of 1000 files of 1MB in size each.
  - **R_10MBx1** – This dataset consists of a single file of 10MB in size.
  - **R_100MBx1** – This dataset consists of a single file of 100MB in size.
  - **R_0-1GBx1,50MB** – This dataset consists of files with size ranging from 0 bytes to 1GB. There is one file for each size, which is always a multiple of 50MB.

[110] The synthetic dataset tool was developed as part of this work and it is available at `https://github.com/stacs-srg/synthetic_datasets` [last access on 15/05/2018].

  - **R_0-1GBx1,100MB** – Same as *R_0-1GBx1,50MB*, but with files of size that are multiple of 100MB.

- **Text dataset T_1MBx1000**. This datasets consists of 1000 files containing English words. The files are generated using the *synthetic dataset tool*, such that the words are chosen at random from a set of 400K+ words. This dataset is used in experiments where there are context predicates that classify assets based on specific words. For the purpose of these experiments, the two words *the* and *bibliopole* were placed in the dataset with an arbitrary probability. The word *the* was placed as a common word in the text, with a probability of p=1/15, while the word *bibliopole* was placed as a rare word, with a probability of $p=1/10^5$. Each file is 1MB in size.

- **Mixed content dataset**. This dataset consists of 1000 files, of which 754 are text files of 1MB each, for a total of 750MB, and 246 are JPEG images[111], for a total of 270MB. The total size of the dataset is 1GB.

8.5 Reference Implementation Benchmarks

This section presents a first set of experiments aimed at testing and understanding the performance and behaviour of the SOS node reference implementation in terms of input/output (IO) and network performance.

8.5.1 Input/Output Benchmark

Writing and reading data are the two most basic operations of any storage systems. In the SOS, IO operations are performed when writing and reading atoms, and manifests, to and from the local storage. The results of the experiments reported in this section define the baseline performance for the prototype presented in Chapter 6. The IO benchmark presented in this section is necessary to understand the results of all other experiments discussed in the remainder of the chapter.

[111] The images were retrieved from the **INRIA Holidays dataset**, available at `http://lear.inrialpes.fr/people/jegou/data.php` [last access on 07/05/2018]. This dataset was chosen because of its variety of content and open license. INRIA is the copyright holder of all the images included in the dataset.

The first experiment consists of adding and retrieving atoms of varying size through the storage management service (SMS) of the SOS node. The results are compared against the write/read performance of the Java standard library *java.io* with the local file system. The SMS writes and reads data through the *castore* library, which itself relies on the *java.io* library. The time to compute the GUID of the atoms is also recorded, so that it is possible to better understand its contribution when adding atoms to a node.

The experiment was run over one single node of the BWC and repeated 10 times. The following variables were used:

- **Independent variables**: Amount of data read/written.
- **Dependent variables**: (1) Time to write the data; (2) time to read the written data; (3) time to add the data as an atom; (4) time to calculate the hash when adding the atom; and (5) time to read the added atom.
- **Controlled variables**: R_0-1GBx1,50MB dataset.

Figure 8.2 shows the IO performance of the SOS when writing/reading atoms. The writing/reading performance of the *java.io* library is also shown (FS Write and FS Read). The time to write/read data always increases in relation to the amount of data. However, the increase is not always linear, as shown from the throughput graph (Figure 8.2[left]). The SOS read atom operation is slower than the file system read one. This is due to the SOS node performing atom look-up operations through the atom manifest before retrieving it, while the file system reads the data directly from disk. However, as shown in the Figure 8.2[right], the loss in performance is reasonably contained. The add atom operation, on the other hand, is much slower than the file system write due to the hashing of the data to generate the GUID of the atom. Thus, the hashing operation defines an upper bound when persisting atoms. Different cryptographic hash functions are benchmarked in the next subsection.

The throughput of the file system write operation drops considerably for files large than 750MB. Running the experiment multiple times has shown that this non-linear behaviour

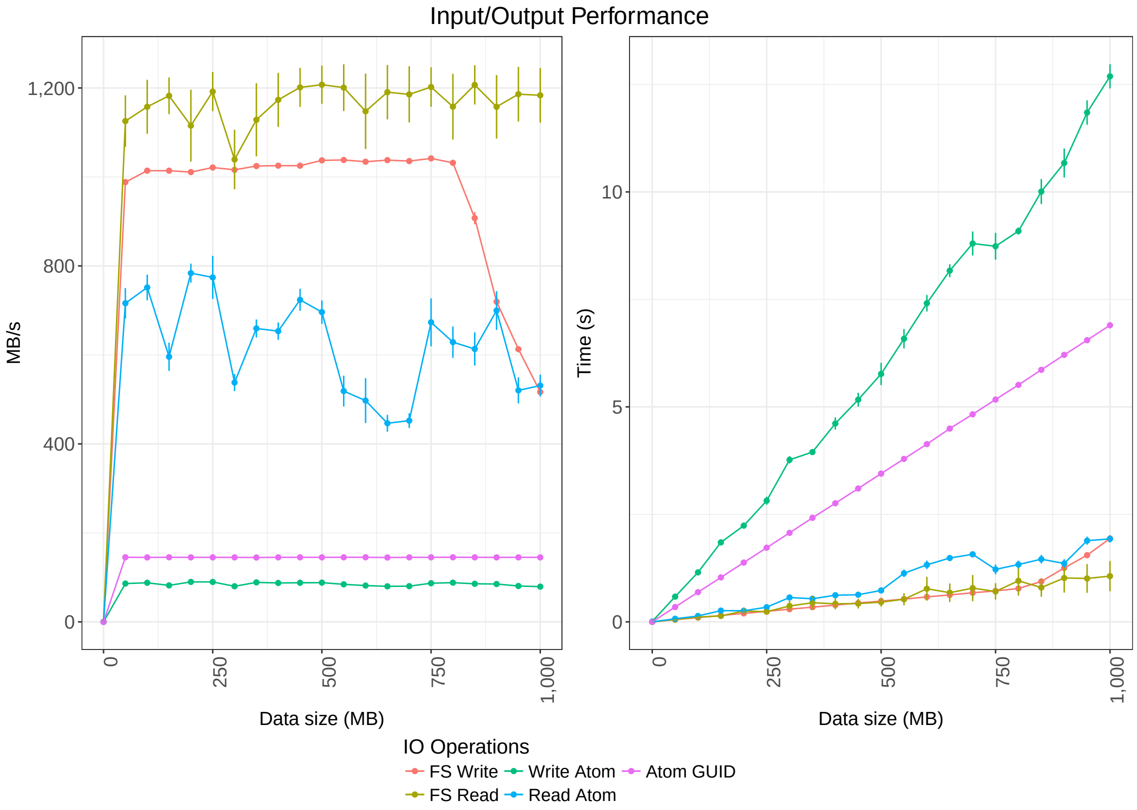


Figure 8.2: Experiment IO. (left) throughput in MB/s – the higher the better; (right) time to perform read/write operations – the lower the better. A 95% confidence interval is shown in each plot.

is correlated to the JVM maximum heap size and garbage collector strategy used.

8.5.1.1 GUID - Hash Function

The experiment presented in this section benchmarks the following cryptographic hash functions: MD5, SHA-1, SHA-256, and SHA-512. The MD5 and SHA-1 hash functions have been benchmarked because of their historical significance, but their usage in distributed systems is progressively decreasing. The current prototype of the SOS relies on the Apache commons-code library[112] for hashing data, so the following results reflect the performance of this library.

The experiment was run over one single node of the BWC and repeated 10 times. The following variables were used:

[112] Apache Commons Codec. https://commons.apache.org/proper/commons-codec/ [last accessed on 30/04/2018].

- **Independent variables**: (1) Amount of data read/written and (2) the hash function used: MD5, SHA-1, SHA-256, and SHA-512.
- **Dependent variables**: Time to hash the data.
- **Controlled variables**: R_0-1GBx1,100MB dataset.

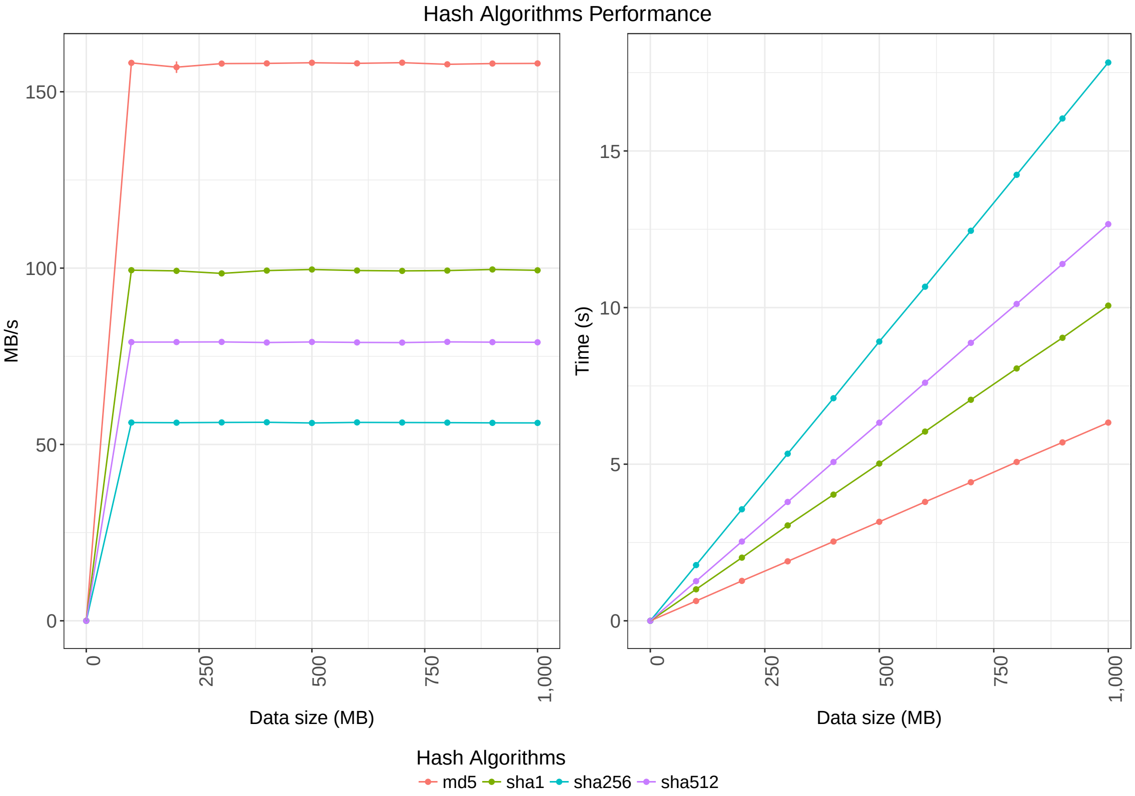


Figure 8.3: Hash algorithms performance. The diagram on the left shows the throughput in MB/s – the higher the better, while on the right is shown the time to run the hash functions – the lower the better. 95% confidence intervals have been calculated and are shown as very small bars, but they hard to see due to the highly consistent performance of the hash functions.

The results outlined in Figure 8.3 show that for each hash function its throughput, expressed in MB/s, is constant regardless of the amount of data processed. The MD5 and SHA-1 hash functions have the highest throughput, at an average of 156.30MB/s (ci= ± 1.46) and 97.99MB/s (ci= ±0.92), respectively. The throughput of SHA-256 is considerably lower, at 55.18MB/s (ci= ±0.52). Unexpectedly, the performance of SHA-512 is better than SHA-256 (77.84MB/s (ci= ±0.73)). This is because SHA-512 relies on 64-bit data structures, which make the hash function run faster on 64-bit processors. However, SHA-256 was still chosen to be the default hash function in the reference implementation because of its widely spread usage elsewhere.

8.5.2 Network Latency and Throughput

Another important aspect of a distributed storage system is its ability to communicate and transfer data to/from other nodes efficiently. The majority of the experiments presented in the remainder of this chapter involve multiple nodes interacting with one another. This experiment benchmarks the throughput between two SOS nodes when run on the BWC.

The experiment was run over two nodes of the BWC and repeated 10 times. Each repetition involved transfers of data of 100MB in size (*R_100MBx1* dataset). Table 8.1 shows that the throughput between two SOS nodes is about 50MB/s, for unsigned requests, which is about 45% the maximum throughput between two nodes of the cluster.[113] The lower throughput is mostly due to the fact that the SOS protocol is built on top of HTTP. Relying on a well-established and tested protocol, such as HTTP, is an advantage, but it affects also the network performance of the node significantly. No significant difference was found when performing the requests between nodes running on different racks. The throughput for signed requests is about 17MB/s, which is about 14% the maximum throughput between two nodes of the cluster. This decrease in throughput is due to the request, and its body, having to be hashed when generating the digital signature (see Section 2.1.8.4). Request signing is disabled for the experiments presented in the remainder of this chapter.

	SOS (MB/s)	SOS - Signed (MB/s)	TCP (MB/s)	UDP (MB/s)
Same Rack	52.8 (ci= ±2.2)	16.5 (ci= ±0.5)	117.9 (ci= ±0.1)	119.6 (ci= ±0.1)
Different Racks	53.4 (ci= ±1.6)	16.8 (ci= ±0.7)	117.9 (ci= ±0.1)	119.6 (ci= ±0.1)

Table 8.1: Throughput between nodes of the Butts Wynd Cluster.

[113] The TCP and UDP throughput between nodes was measured using iperf (v. 2.0.10), a network performance utility available at `https://iperf.fr/` [last accessed on 18/05/2018].

8.5.3 Atoms and Manifests Replication

The SOS provides resiliency by performing data replication across multiple nodes. In the reference implementation presented in this work, replication of content — atoms and manifests — is not available by default. However, users can define contexts that capture a particular set of the managed content within a node and replicate it over a codomain of nodes. Prior to presenting the context related experiments (see Section 8.6), we investigate the cost of replicating atoms and manifests and whether there is any benefit in performing the replication process in parallel, when the replication factor is greater than one.

- **Independent variables**: The replication factor.
- **Dependent variables**: Time to replicate content such that the replication factor is satisfied.
- **Controlled variables**: The *R_1MBx1*, *R_10MBx1*, *R_100MBx1* datasets were used. Versions manifests were used when replicating manifests.

The diagrams in Figure 8.4 show that the cost of replicating content increases linearly with respect to the replication factor, as expected. However, replicating content over multiple nodes in parallel can significantly decrease the overall replication time, such that almost no increase in cost is observed as the replication factor increases. The bandwidth consumption increases in relation to the level of parallelism. The results show that parallel replication has a lower cost than linear replication, but only for replication factors greater than 2 and 3. The overhead of creating and managing threads within a thread pool, in fact, might be counterproductive when there are few replications to perform. The behaviour is consistent whether the replicated content are atoms of different size or manifests (between 100 bytes and 1KB in size).

The experiments presented below use sequential atom/manifest replication. Upon reading and evaluating these results, however, one should take into account the possibility of replicating content in parallel and the advantages that can be gained from it.

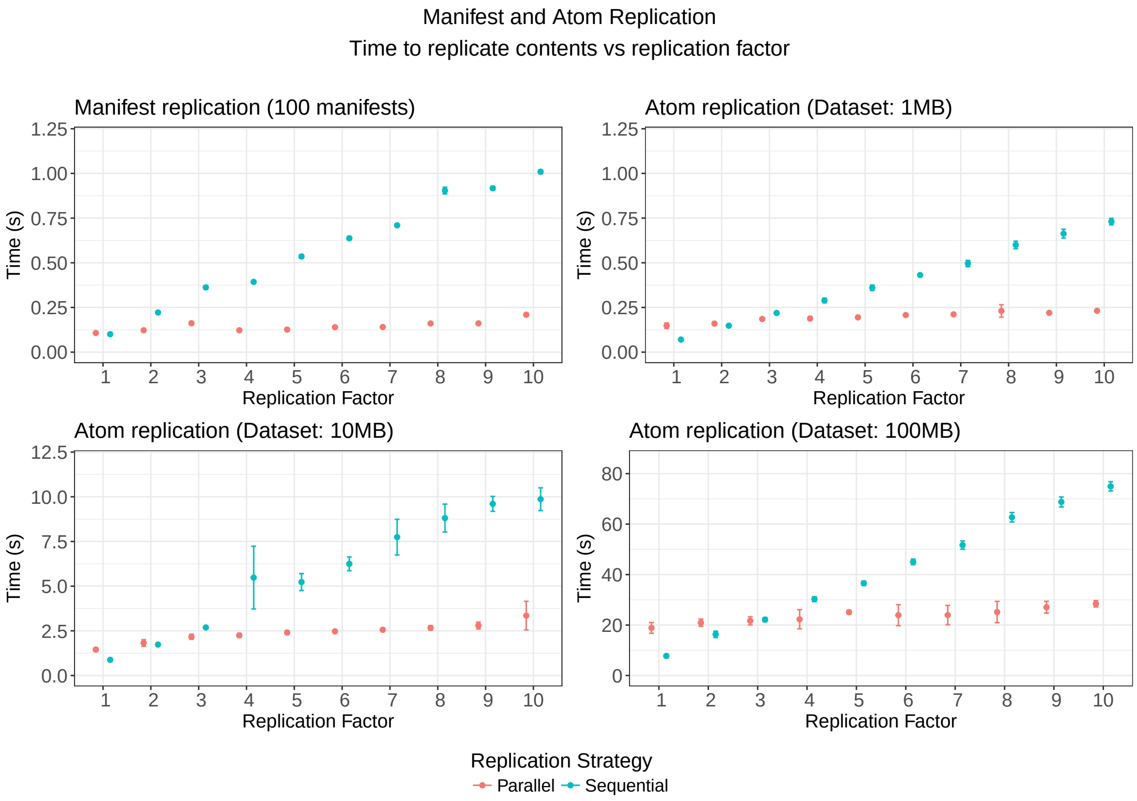


Figure 8.4: Replications of Manifests and Atoms of different sizes. Lower time values are better.

8.6 Context Experiments

The context model is the component of the SOS that allows users to define rules to automatically manage content within a set of nodes, as described in Section 5.1.7. The different elements of the context model (*i.e.*, the predicate, domain, policies, and codomain) are evaluated individually and in conjunction, with the aim to understand how each of them affects the time to run a context and therefore the viability of using contexts in a small distributed environment. Observations on how contexts react to nodes failures were also made and reported.

The following experiments were run:

- **The Nature of the Predicates** – Section 8.6.1. This experiment demonstrates how the nature of the predicate affects the time taken to identify contents for a given context. The results show that predicates that operate on manifests only have a significant lower cost than predicates operating on atoms. Predicates, however, can be optimised by filtering atoms to be inspected by looking at their associated metadata first.

- **Predicates and the Domain** – Section 8.6.2. These experiments describe the behaviour and performance of contexts that are run over a domain consisting of more than one node. The experiments show that distributing the workload across multiple nodes, of the domain, can improve the overall time to run the predicate for small domains.
- **The Nature of the Policies** – Section 8.6.3. These experiments explore the nature of the policies and how they affect the overall running time of the contexts to which they belong.
- **Policies and the Codomain** – Section 8.6.4. These experiments show that the cost of enforcing policies over a codomain depends on the number of nodes over which the policy applies.
- **Context under Failure** – Section 8.6.5. These experiments illustrate the behaviour of contexts when some of the nodes in the domain or codomain fail. In particular, these experiments show that contexts can be used to automatically maintain content in the SOS under failure.

8.6.1 The Nature of the Predicates

The predicate is a boolean-valued function that operates over SOS content (see Section 5.1.7.2). The experiment presented in this section shows that the intrinsic nature of the predicate and the type of content it inspects have a significant effect on its running time. The first part of this experiment evaluates predicates that identify assets based on their metadata and content, which are all of textual type. The second part of this experiment attempts to simulate a more realistic scenario, where data is of multiple data types and predicates classify content based on its extrinsic and intrinsic metadata (see Section 2.1.3 for the different types of metadata).

This experiment was run over a single node of the BWC and repeated 10 times. The following variables were used:

- **Independent Variables**: The predicate functions, as listed below.
- **Dependent Variables**: The time to run the predicate for the entire dataset.
- **Controlled Variables**: The *T_1MBx1000* dataset, which consists of text files only.

The predicates used for this experiment classify assets based on their content and metadata related to them. The choice of the predicates was driven by the dataset used and resulted in the following list:

- **Base** – Base Predicate. This predicate returns true irrespective of the content processed.
- **D_CWOO** – Data predicate, common word occurs once. This predicate returns true if the word *the* is found in the atom.
- **D_UWOO** – Data predicate, uncommon word occurs once. This predicate returns true if the word *bibliopole* is found in the atom.
- **D_CWO10** – Data predicate, common word occurs at least 10 times. This predicate returns true if the word *the* is found at least 10 times in the atom.
- **MD_CWOO** – Metadata and data predicate, common word occurs once. This predicate returns true if the atom is of *text* type and the word *the* is found in the atom.
- **MD_UWOO** – Metadata and data predicate, uncommon word occurs once. This predicate returns true if the atom is of *text* type and the word *bibliopole* is found in the atom.
- **MD_CWO10** – Metadata and data predicate, common word occurs at least 10 times. This predicate returns true if the atom is of *text* type and the word *the* is found at least 10 times in the atom.

- **Metadata**. This predicate returns true if the version's content is associated with metadata that has the Content-Type attribute equal to *text*.

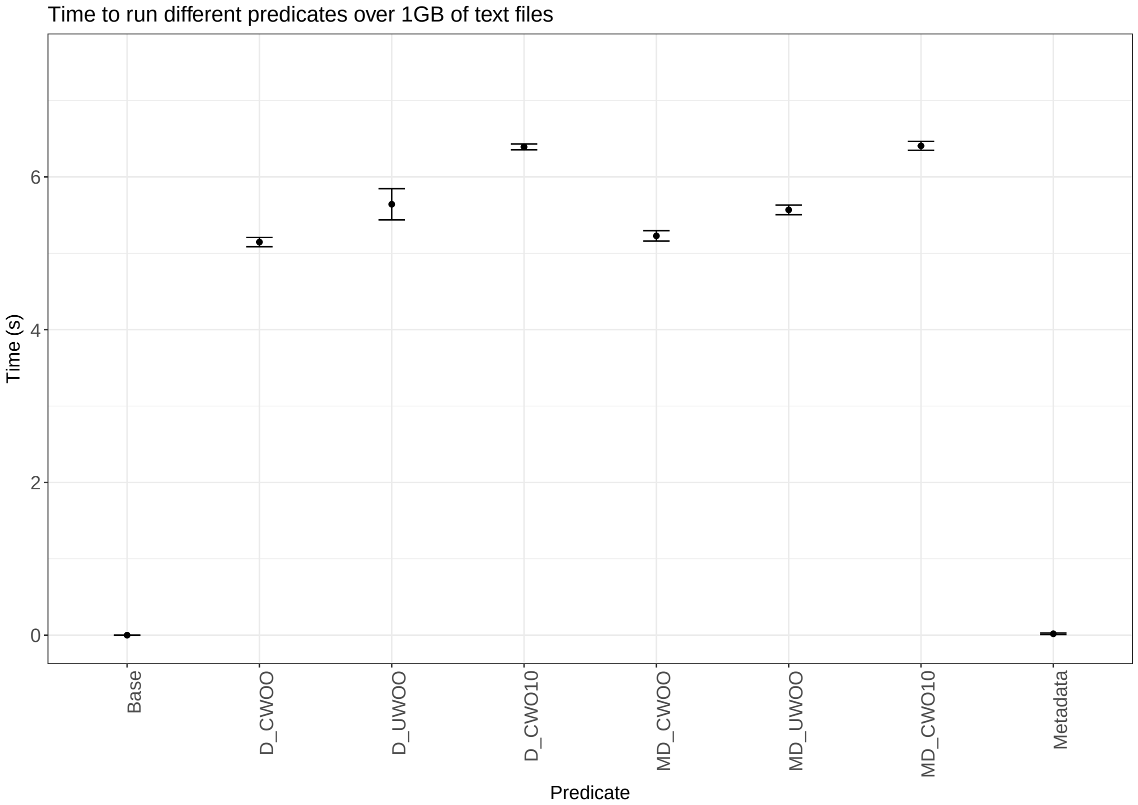


Figure 8.5: Time needed to run different types of predicates over 1000 text files of 1MB each. Lower time values are better.

Figure 8.5 shows that the predicates operating on the atoms — *D_CWOO*, *D_UWOO*, *D_CWO10*, *MD_CWOO*, *MD_UWOO*, *MD_CWO10* — are the most expensive, due to the predicate having to process a significant amount of bytes before being able to return. The predicates searching for the matching word only once — *D_CWOO*, *D_CWO10*, *MD_CWOO* and *MD_UWOO* — return earlier, while the predicates *D_CWO10* and *MD_CWO10* are more expensive since they have to process the atoms until ten occurrences of the matching word are found. Likewise, predicates searching for the uncommon word *bibliopole* are more expensive than predicates searching for the more common word *the*. For this experiment, all files of the dataset are of *text* type, thus processing metadata and atoms together — *MD_CWOO*, *MD_UWOO*, and *MD_CWO10* — has no effect, since the metadata condition will always return true. On the other hand, the *Metadata* predicate has costs closer to the *Base* predicate.

The conclusion that can be drawn from these results is that the feasibility of running a predicate within a reasonable amount of time depends on (1) the type of computation run and (2) the type of content being processed. This experiment, however, does not take into account the possibility of running the predicate functions in parallel, which could improve significantly the amount of time to process the entire dataset.

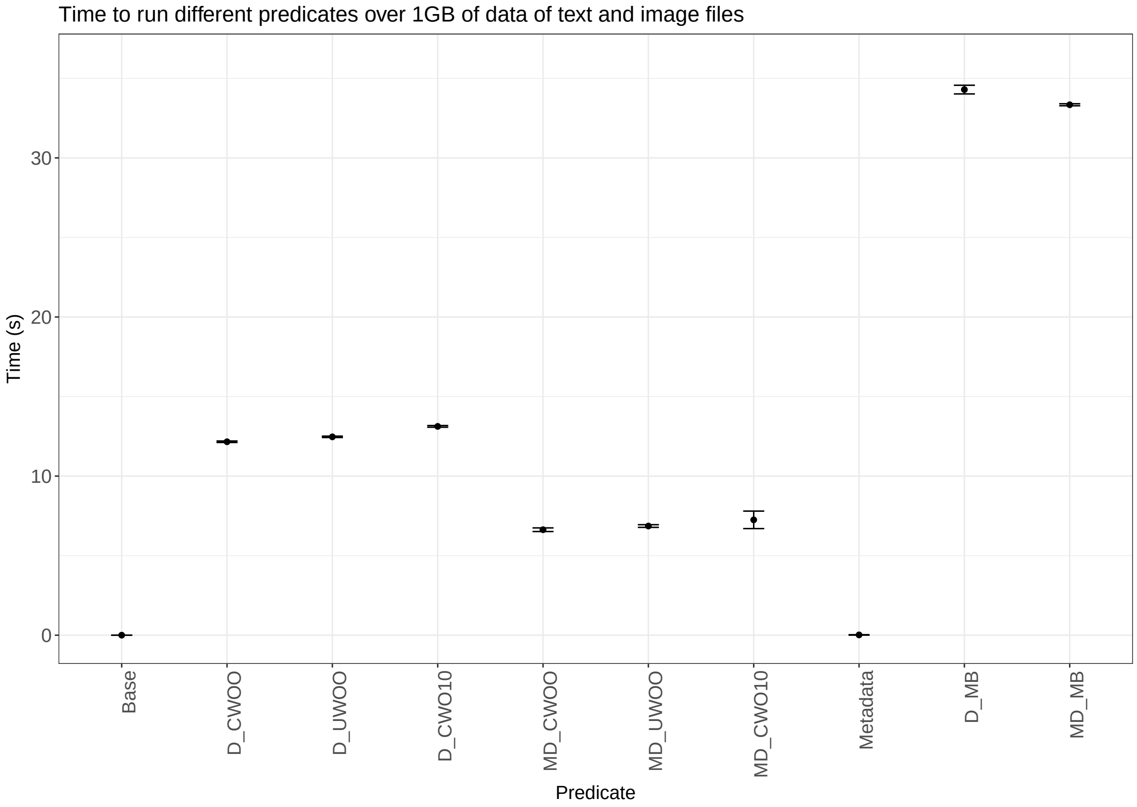


Figure 8.6: Time needed to run different types of predicates over the mixed content dataset. Lower time values are better.

In a real world scenario a dataset can contain more diverse data, in terms of data size and type. For the following experimental instance, the *Mixed Content* dataset was used (see Section 8.4.3). The experiment was run using the same settings of the previous instance, but using two additional predicates:

- **D_MB** – Data predicate, mostly blue images. This predicate returns true if the atom is a JPEG and is mostly blue (higher prevalence of blue colour among the RGB values).

- **MD_MB** – Metadata and data predicate, mostly blue images. This predicate returns true if the version's content is a JPEG image, based on the metadata, and is mostly blue.

The results shown in Figure 8.6 demonstrate how running the predicate over data as well as metadata can result in time savings. For the case of the predicates searching text within the data, the overall running time of the predicates is 30% lower when the assets are filtered based on their metadata. This improvement is due to the fact that the MD_CWOO, MD_UWOO, and MD_CWO10 predicates filter out about 30% of the dataset, which are the images. A similar behaviour is observed for the *D_MB* and *MD_MB* predicates.

The conclusion that can be derived from this second run of the experiment is that the cost of a predicate depends on whether it processes manifests, atoms, or a combination of both. The cost of running a predicate over an atom is directly proportional to the size of the atom, but it also depends on the actual computation run. Searching a word in an atom is inherently cheaper than calculating the most prevalent RGB colour of an image. Having predicates using machine learning algorithms to identify content, as hinted in Section 5.1.7.2, allows users to express richer computation, but at a significantly higher cost. Therefore, when writing predicates one should take into account the inherent nature of the computation to be run and whether it is possible to filter out part of the content based on its metadata.

8.6.2 Predicates and the Domain

This section investigates the behaviour and performance of predicates that are run over a domain consisting of one or more nodes. Three experiments were run in relation to:

- The **node-cardinality** of the domain - Section 8.6.2.1. This experiment studies the relationship between the number of nodes in the domain and the cost of running a context's predicate over them.
- The **data-cardinality** of the domain – Section 8.6.2.2. This experiment studies the relationship between the total number of assets in the domain and the cost of running a context's predicate over them.
- The total **amount of data** stored within the domain – Section 8.6.2.3. This experiment studies the relationship between the total amount of data in the domain and the cost of running a context's predicate over it.

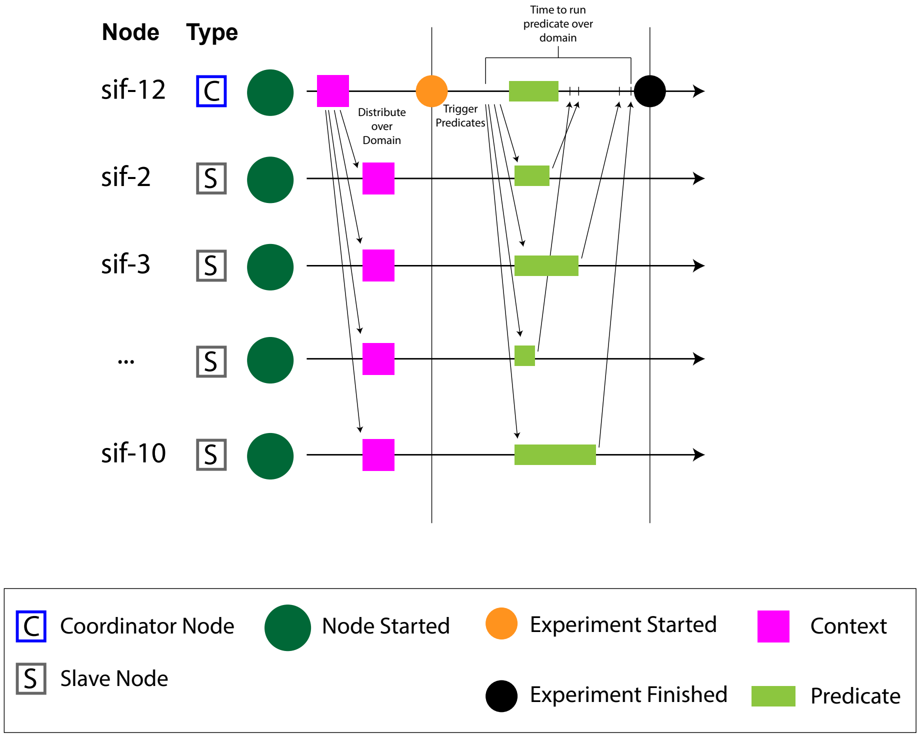


Figure 8.7: Diagram showing the interaction between the nodes involved when evaluating a predicate run over a domain of multiple nodes. This diagram is not shown in scale.

The diagram in Figure 8.7 illustrates the steps performed at each iteration of the experiments. First of all, the dataset is distributed evenly across the domain, prior to starting the nodes. Then, the nodes are started and the context initially added on the coordinator node is distributed to all the nodes of the context's domain. Thereafter, the experiment is started and the predicates on the slave nodes are manually triggered from the coordinator nodes (in the non-experimental setup, predicates are scheduled to run periodically). A node acknowledges the coordinator as soon as it finishes running the predicate. The time between triggering the predicates and all the nodes returning back to the coordinator is recorded. Finally, the experiment iteration terminates.

8.6.2.1 Variation over the Node-Cardinality of the Domain

This experiment investigates the behaviour of a context when its predicate is run over a domain consisting of one or more nodes. The dataset used for the experiment is distributed evenly across the domain, which can consist of one (coordinator node only) up to ten nodes. The experiment was run over ten nodes of the BWC and repeated 10 times. The following variables were used:

- **Independent Variables**: The number of nodes in the domain.
- **Dependent Variables**: The time to run the predicate for the entire dataset over all the nodes in the domain.
- **Controlled Variables**: The *D_UWOO* predicate and the *T_1MBx1000* dataset.

The results shown in Figure 8.8 indicate that distributing the computation of the predicate across multiple nodes decreases the overall running time to evaluate a context's predicate. The optimal solution was found for domains of three nodes. Running the predicate over domains greater than three, however, still takes less time than running the predicate in one single node, because the overall amount of data to be processed by one node is less and run in parallel across the other nodes of the domain.

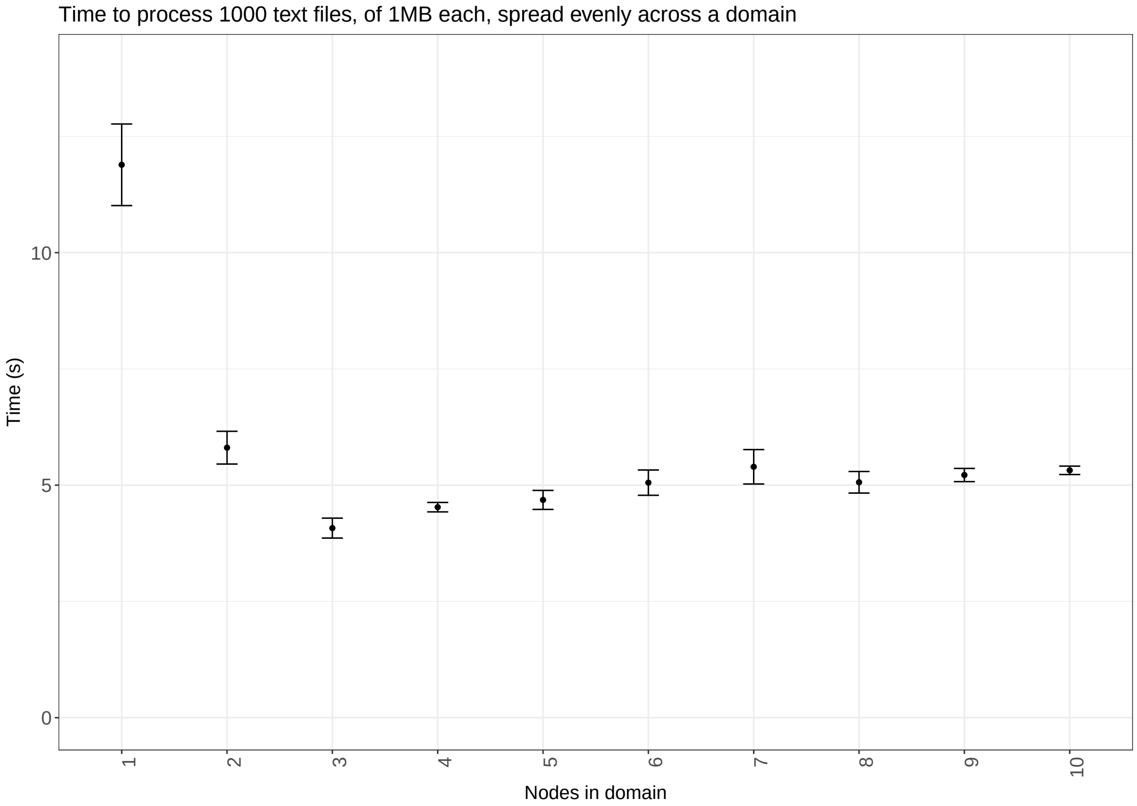


Figure 8.8: Variation over the node-cardinality of the domain.

8.6.2.2 Variation over the Data-Cardinality of the Domain

This experiment investigates the behaviour of the context when the number of nodes in the domain is fixed, but the number of assets to process changes. For this experiment, domains of one, two, three, six, and ten nodes were chosen, while the predicate was run over an increasing number of assets (distributed evenly across the domain). The experiment was run over ten nodes of the BWC and repeated 10 times for each node-cardinality of the domain and data point in the x-axis. The following variables were used:

- **Independent Variables**: The overall number of files spread across the domain.
- **Dependent Variables**: The time to run the predicate for the entire dataset over all the nodes in the domain.
- **Controlled Variables**: The *D_UWOO* predicate and the *T_1MBx1000* dataset.

Domains of one, two, three, six, and ten nodes were chosen.

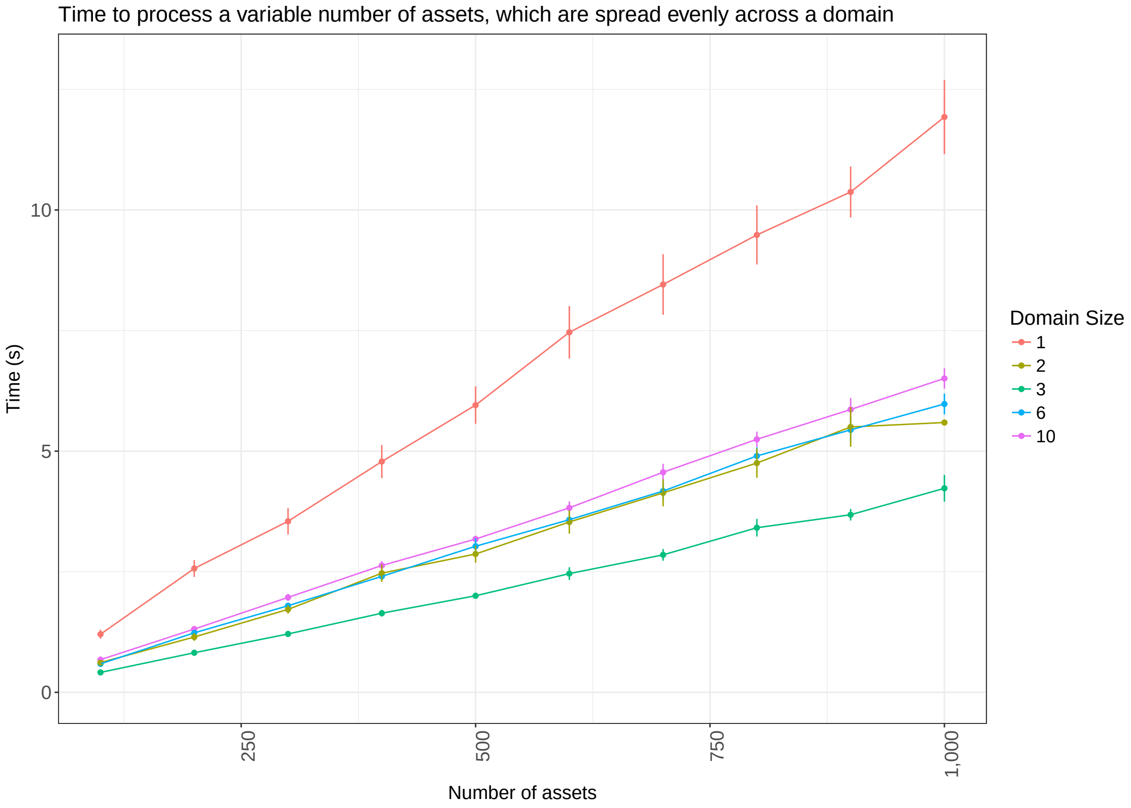


Figure 8.9: Variation over the data-cardinality of the domain. Lower time values are better.

The results in Figure 8.9 show that as the number of assets to process increases, so does the time to run the predicate over all the content stored in the domain. This behaviour is consistent, irrespective of the node-cardinality of the domain. These results also confirm that the optimal number of nodes in a domain is three. This is due to the overhead of having to distribute and coordinate the computation across multiple nodes. The overhead might become negligible for certain types of predicates and amounts of data to process.

8.6.2.3 Variation over the Amount of Data Stored within the Domain

Finally, this experiment investigates how the time to run a predicate over the domain is affected by the overall amount of data to process. The data to be processed by the predicate was distributed evenly across the domain before the start of each experimental iteration. This experiment was run over six nodes in the BWC and repeated 10 times for each dataset size. The following variables were used:

- **Independent Variables**: The amount of data to process, which ranged from 100MB to 1GB, at intervals of 100MB.
- **Dependent Variables**: The time to run the predicate for the entire dataset over the domain.
- **Controlled Variables**: The *D_UWOO* predicate and the *T_1MBx1000* dataset. A domain of six nodes was chosen.

Figure 8.10 shows that the cost of running the predicate over the domain is directly proportional to the amount of data to process, as expected.

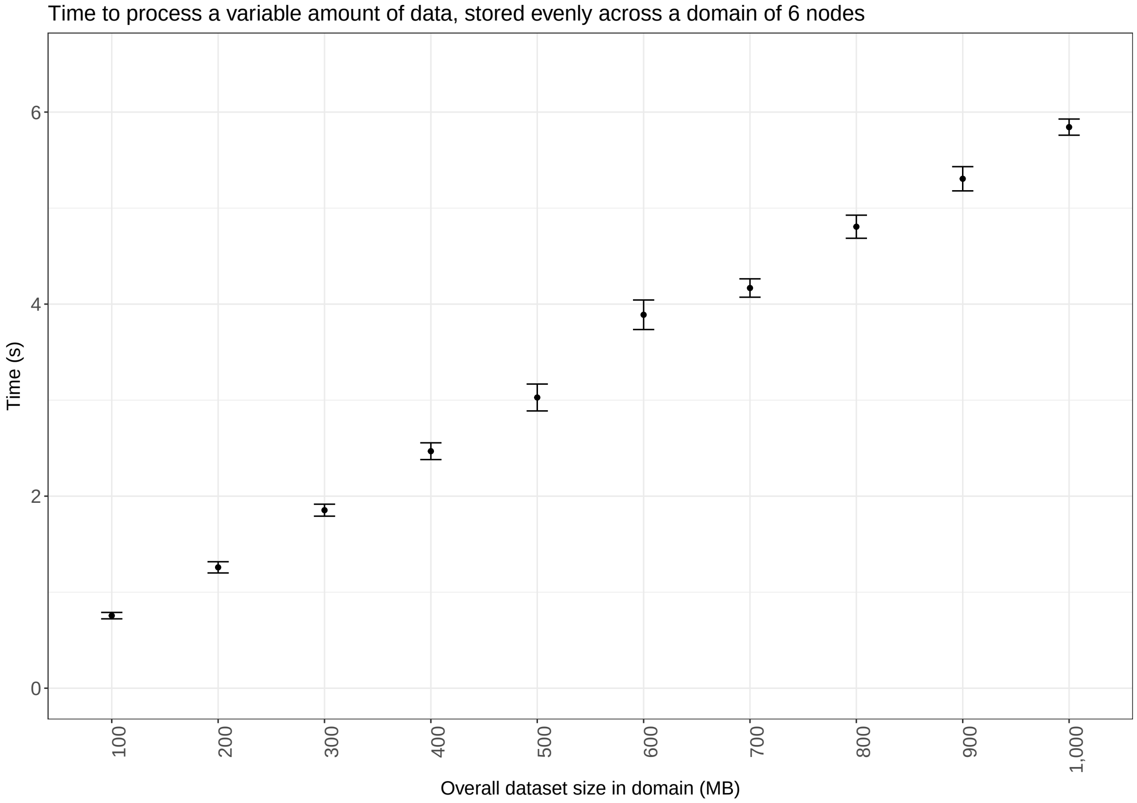


Figure 8.10: Time to run the predicate over a domain of nodes.

8.6.3 The Nature of the Policies

This section studies the relationship between the nature of a policy and the time it takes to enforce it. The following context policies were evaluated:

- **Void Policy**. The context has one policy, which does not do any computation. The result for this policy defines the baseline for all the other policies.
- **Grant Access**. The context has one policy which considers atoms protected by one role and extends the permissions to another role. The policy does not interact with other nodes. Keys, in this particular instance, are encrypted using 2048-bit RSA asymmetric keys.
- **Atom Replication**. The context has one policy which replicates the content of an asset to another node of the codomain.
- **Manifest Replication**. The context has one policy which replicates the asset's version manifest to another node of the codomain. The version's content is not replicated.

For the purpose of this experiment, and the ones following, atoms and manifests are replicated atomically, as explained in Section 2.1.6.1. Thus the space cost of replicating an atom or manifest of size D by a replication factor n is $D \times n$. Better space cost factors can be achieved using different types of erasure coding techniques, as explained in Section 2.1.6.2. The current SOS implementation, however, does not support erasure coding yet, so it could not be tested in this experiment. Similarly, one may construct policies that replicate content over nodes that respect specific criteria, for example based on their geo-location, availability, or reliability.

This experiment was run over two nodes of the BWC and repeated 10 times for each policy. The following variables were used:

- **Independent Variables**: The type of policy to enforce, as listed above

- **Dependent Variables**: The time to enforce the policy *apply procedure* and the policy *satisfy function* for the entire dataset.
- **Controlled Variables**: The *R_1MBx1000* dataset was used.

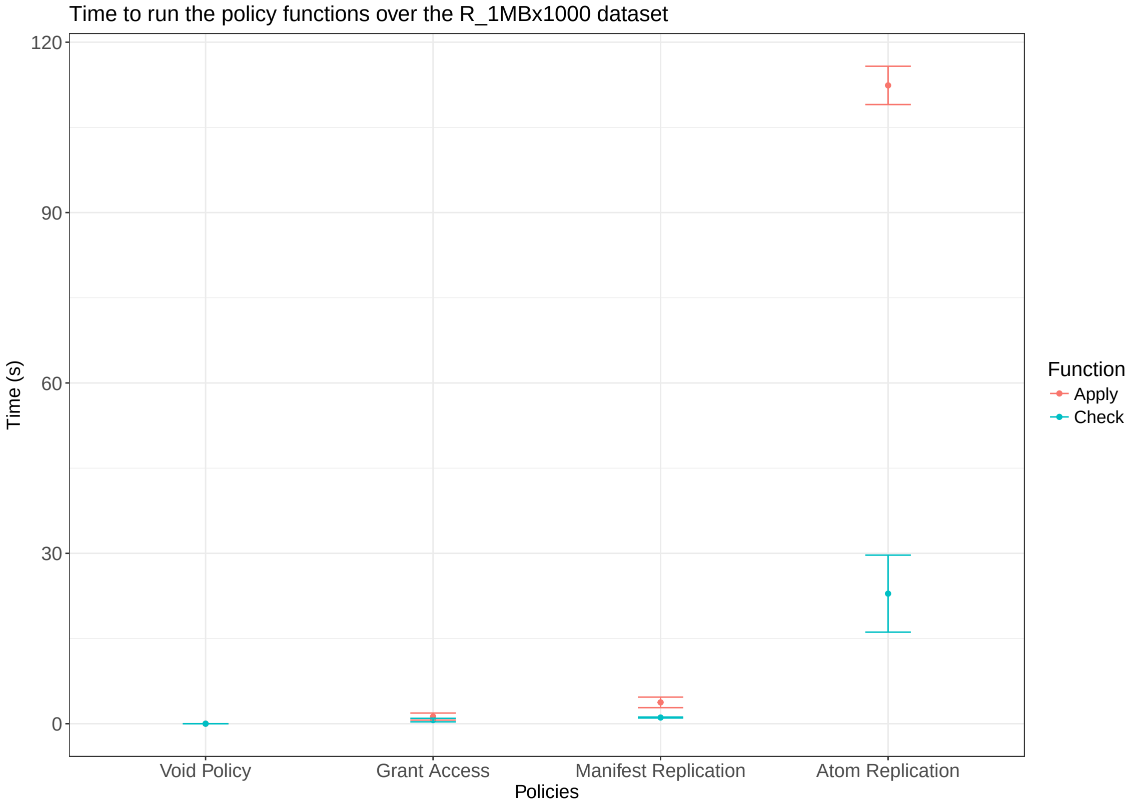


Figure 8.11: Costs of enforcing policies of different types. Lower time values are better.

The results shown in Figure 8.11 indicate that policy *apply procedures* can be significantly more expensive than the policy *check functions*. The time to run the policy *apply procedures* depends on two factors: (1) the amount of content over which it operates and (2) the types of entities that are processed. For instance, policies that operate on atoms tend to be more expensive than policies that operate on manifests only. Policies that operate within the local node are also cheaper than policies that operate within a codomain of nodes.

8.6.4 Policies and the Codomain

This section evaluates the performance of policies when run over codomains consisting of more than one node. For the experiments presented below, the time to enforce an atom replication policy over a given codomain was measured. Two experiments were run. In the first, the replication factor was kept constant, while the number of nodes in the codomain ranged between one and ten (excluding the local node). In the second, the replication factor was equal to the number of nodes in the codomain, which ranged between one and ten nodes (excluding the local node).

The experiments were run over eleven nodes of the BWC and repeated 10 times for each instance. The following variables were used:

- **Independent Variables**:
  - First experiment: The number of nodes in the codomain increases.
  - Second experiment: The number of nodes in the codomain increases. The replication factor also changes and is always equal to the number of nodes in the codomain.
- **Dependent Variables**: The time to enforce the policy *apply procedure* and the policy *satisfy function*.
- **Controlled Variables**:
  - First experiment: The replication factor was set to 1. The *R_1MBx1000* dataset was used. The replication was performed sequentially (see replication experiment in Section 8.5.3).
  - Second experiment: The *R_1MBx1000* dataset was used. The replication was performed sequentially.

The results shown in Figure 8.12 show that the cost of enforcing a policy depends primarily on the *apply procedure* and the number of nodes over which it must be applied.

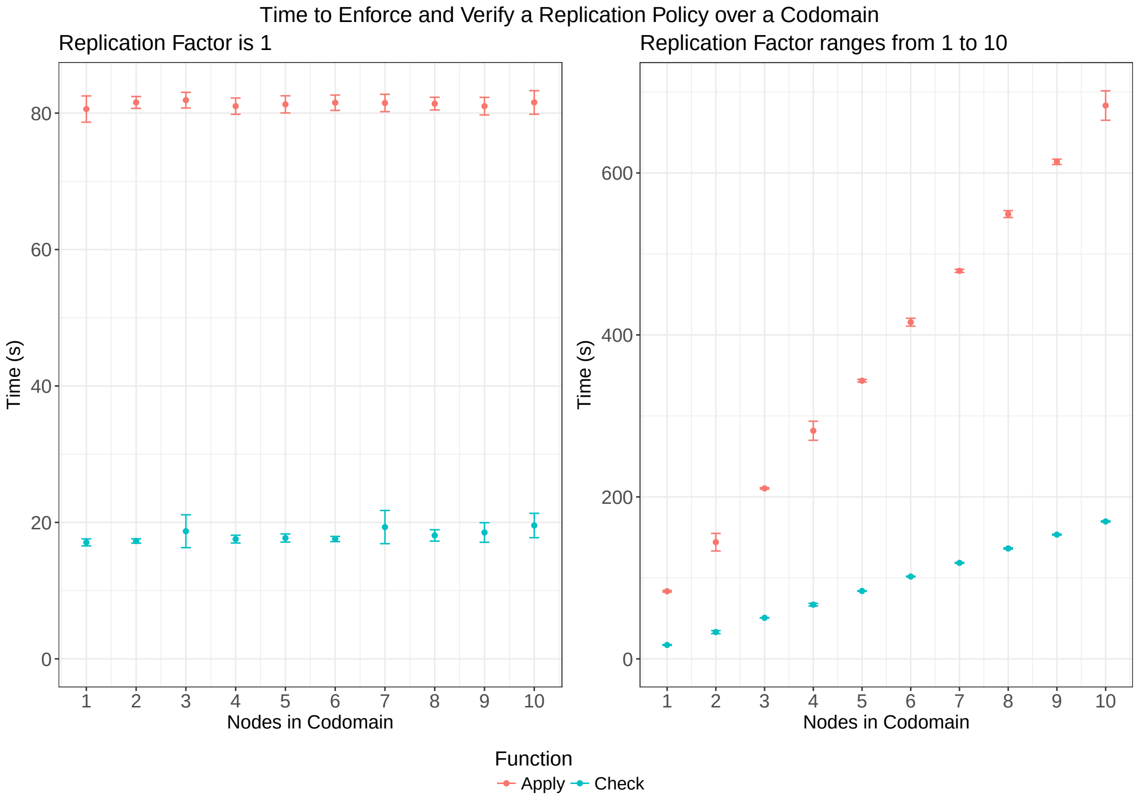


Figure 8.12: Experimental results showing the relationship between the number of nodes in a codomain and the cost to enforce a policy over it. Lower time values are better.

When the replication factor is kept constant (see Figure 8.12[left]), the cost of enforcing the policy also remains constant, irrespective of the number of nodes in the codomain. When the replication factor increases (see Figure 8.12[right]), so does the cost to enforce the policy, both for its *apply procedure* and *check function*.

The conclusion drawn from these results is that the viability of using contexts whose policies enforce changes over a codomain of nodes depends on: (1) the actual rule enforced, as shown in the previous experiment, and (2) on the number of nodes that the policy interacts with. The cost of running a policy that interacts with multiple nodes can be improved significantly by applying the changes in parallel, as previously shown in Section 8.5.3.

8.6.5 Contexts under Failure

This section studies the behaviour of content replication contexts when nodes in the codomain fail. For the purpose of this study the predicates and policies are invoked manually through the experimental framework, rather than periodically scheduled by the context management service (CMS), as described in Section 5.1.7.5. Node failure is controlled by disabling/enabling the REST API of the node. In the results presented, failure is represented in terms of the number of assets for which the policy is not satisfied.

Seven failure cases are studied in the remainder of this section. The **dependent variable** measured in these experiments is the percentage of assets for which the context's policy is valid over time. The **independent** and **controlled variables** are different for each experiment. Each experiment was repeated 5 times and run over multiple nodes of the BWC. The results are plotted in line graphs, where each line has its own colour and represents a run of the experiment. The results of all the runs are plotted separately, rather than aggregated, because the purpose of these experiments is to demonstrate the behaviour of contexts rather than studying their quantitative performance.

The experimental design and the observed behaviour are described for each case study. Note that the workflow diagrams are not drawn in scale.

Workload

The workload of the experiments presented in this section consists of contexts that replicate the content identified by their predicates. For the purpose of this experiment, a predicate that returns true for all the assets of a node is used. The policy of the context replicates the content of a given asset n-times, where n is the replication factor.

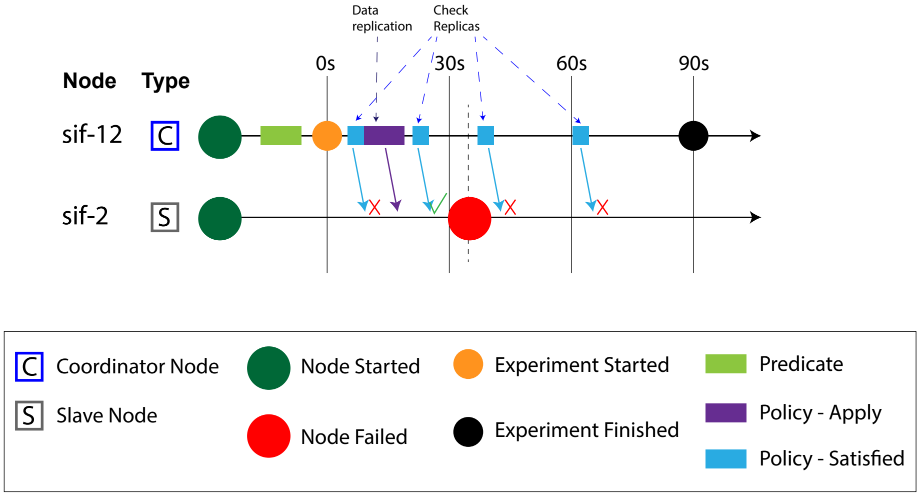
The *R_100KBx100* dataset was used for these experiments. The purpose of these experiments is not to measure how well the reference implementation manages large quantities of data, but rather how contexts react to changes within the codomain.

8.6.5.1 Case 1: The Node Failure

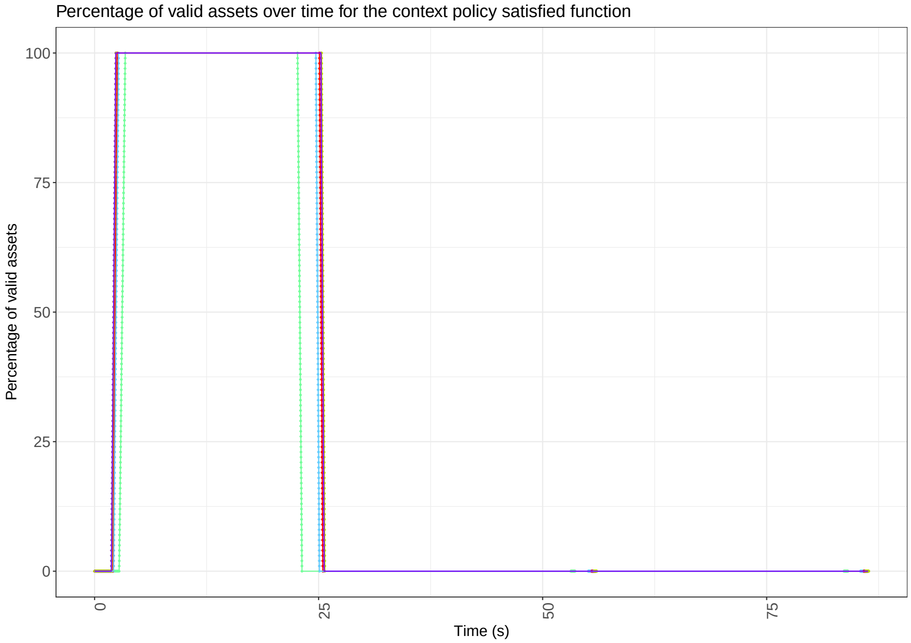
This experiment evaluates the ability of a context to detect the failure of a node in its codomain. The context's policy *satisfied function* detects the failure of the node when verifying that the content replicated by the context's policy is still available.

The experiment was run over two nodes of the BWC, where one node — the coordinator — manages a context that replicates content to the slave node. The diagram in Figure 8.13a shows the actions taking place during the experiment, while Figure 8.13b shows the number of assets for which the policy is valid over time.

First, the policy *satisfied function* of the context is run and detects that no replicas exist in the codomain. Then the policy *apply function* of the context is run and the coordinator replicates the content of the dataset to the slave node. The percentage of valid assets, however, remains zero — as reflected in the first few seconds shown in Figure 8.13b — until the policy *satisfied function* is run again and reaches hundred percent (*i.e.*, all assets are replicated to the slave node). After a few seconds, the slave node fails and the replicas it stored become unavailable. The coordinator detects that the replicas are unavailable only when the policy *satisfied function* is run again, and the percentage of valid assets for the policy decreases back to zero.



(a) Experiment workflow.



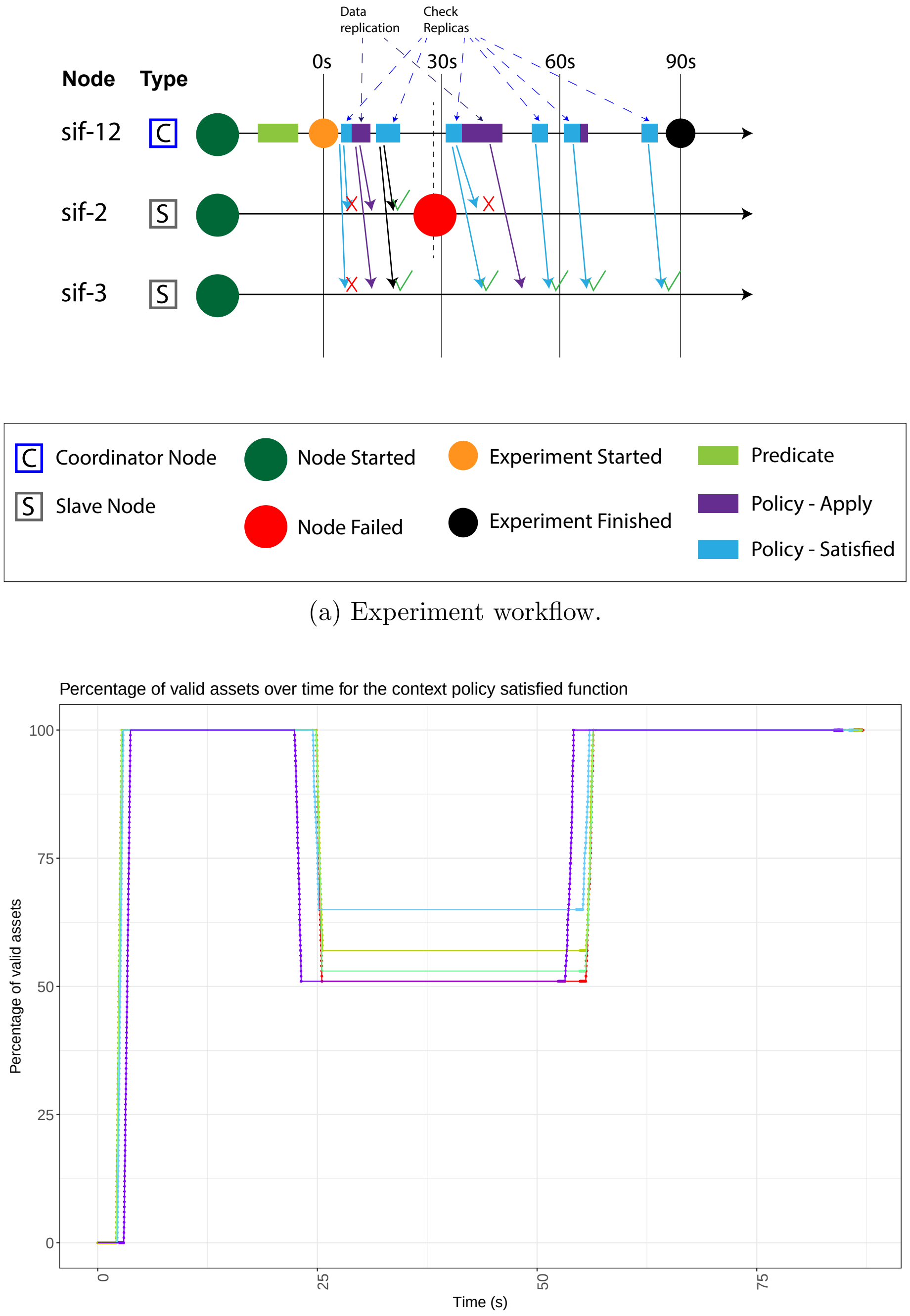
(b) Experiment results.

Figure 8.13: Node failure: the replica node fails and the main node is unable to detect it any more. The different colours used to indicate the multiple runs of the experiment in (b) are not related to the colours used in (a).

8.6.5.2 Case 2: Partial Failure within the Codomain

This experiment evaluates the ability of a context to continue to operate when some but not all the nodes in the codomain fail. The context's policy *satisfied function* detects the failure of the nodes by verifying whether the content is properly replicated on each node. This experiment was run over three nodes of the BWC — one coordinator and two slave nodes (see Figure 8.14a). Of the two slave nodes only one fails, so that the coordinator node can re-establish the wanted state of the SOS by replicating the content to the other slave node of the codomain that remains available.

The initial phase of the experiment is similar to the previous **Case 1** (see above), but the content is replicated to one of the two slave nodes. After a few seconds from the replication, the slave node `sif-2` fails. This failure means that some of the replicas are unavailable and the number of valid assets drops (see Figure 8.14b). The next run of the policy *apply procedure* re-establishes the desired state of the SOS, by replicating to node sif-3 the replicas that became unavailable when sif-2 failed.



(a) Experiment workflow.

(b) Experiment results.

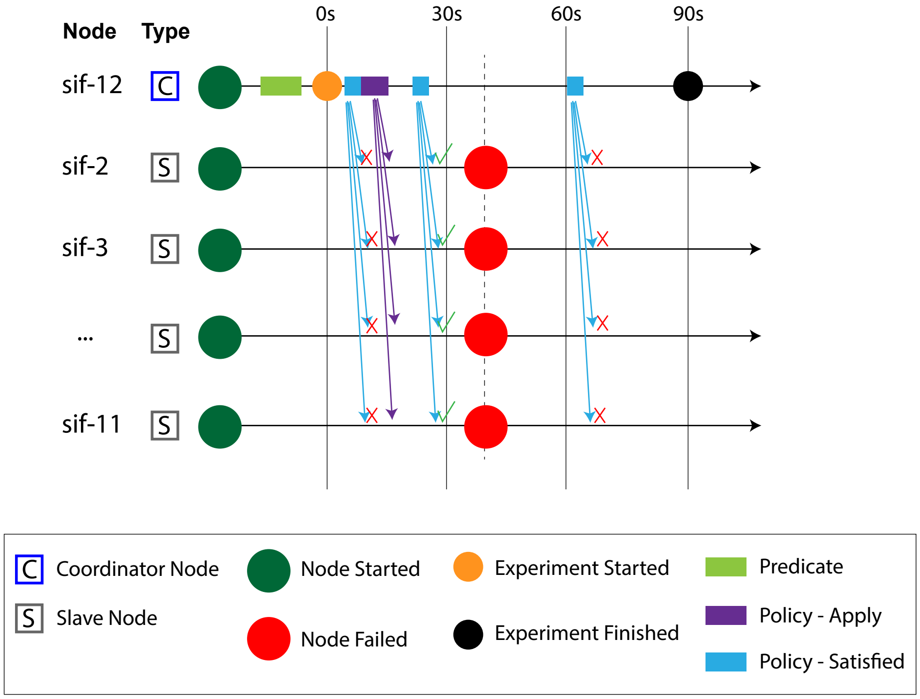
Figure 8.14: Partial failure within the codomain: the replica node fails, but other nodes are available for replication.

This experiment shows that a context can have policies that replicate data within a desired set of nodes and an appropriate replication factor, while node failures are tolerated as long as there are enough nodes to satisfy the policy.

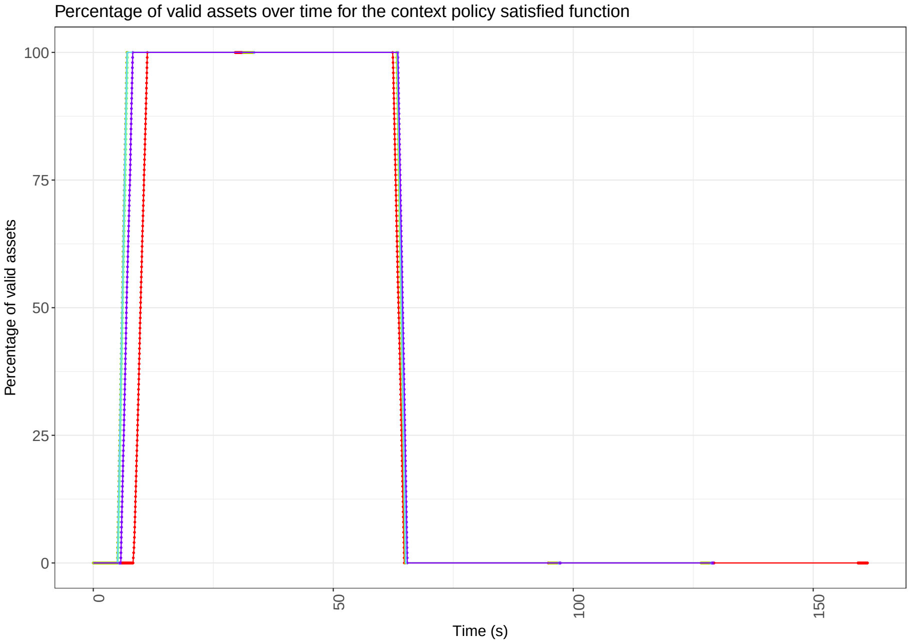
8.6.5.3 Case 3: Failure of all the Nodes in the Codomain

This experiment evaluates the ability of a node to detect the failure of multiple nodes within its codomain. The experiment was run over eleven nodes of the BWC — one coordinator and ten slave nodes (see Figure 8.15a). This experiment is an extension of **Case 1** (see above), but there are ten replica nodes, which all fail at around the same time during the experiment. The purpose of this experiment is to show that contexts are able to detect failures of multiple nodes the same way as the failure of a single node was detected in **Case 1**.

At the beginning of the experiment, the content associated with the context is replicated within the codomain with a replication factor of **3**. Thus, in this instance a value of 100% of valid assets (see Figure 8.15b) means that the content of each asset is replicated to three other nodes, for a total of 300 replicas across the codomain. After few seconds that the coordinator node detects the successful replication of the assets, all the slave nodes fail at about the same time. The failure is recorded by the coordinator as soon as the policy *satisfied function* is run again.



(a) Experiment workflow.



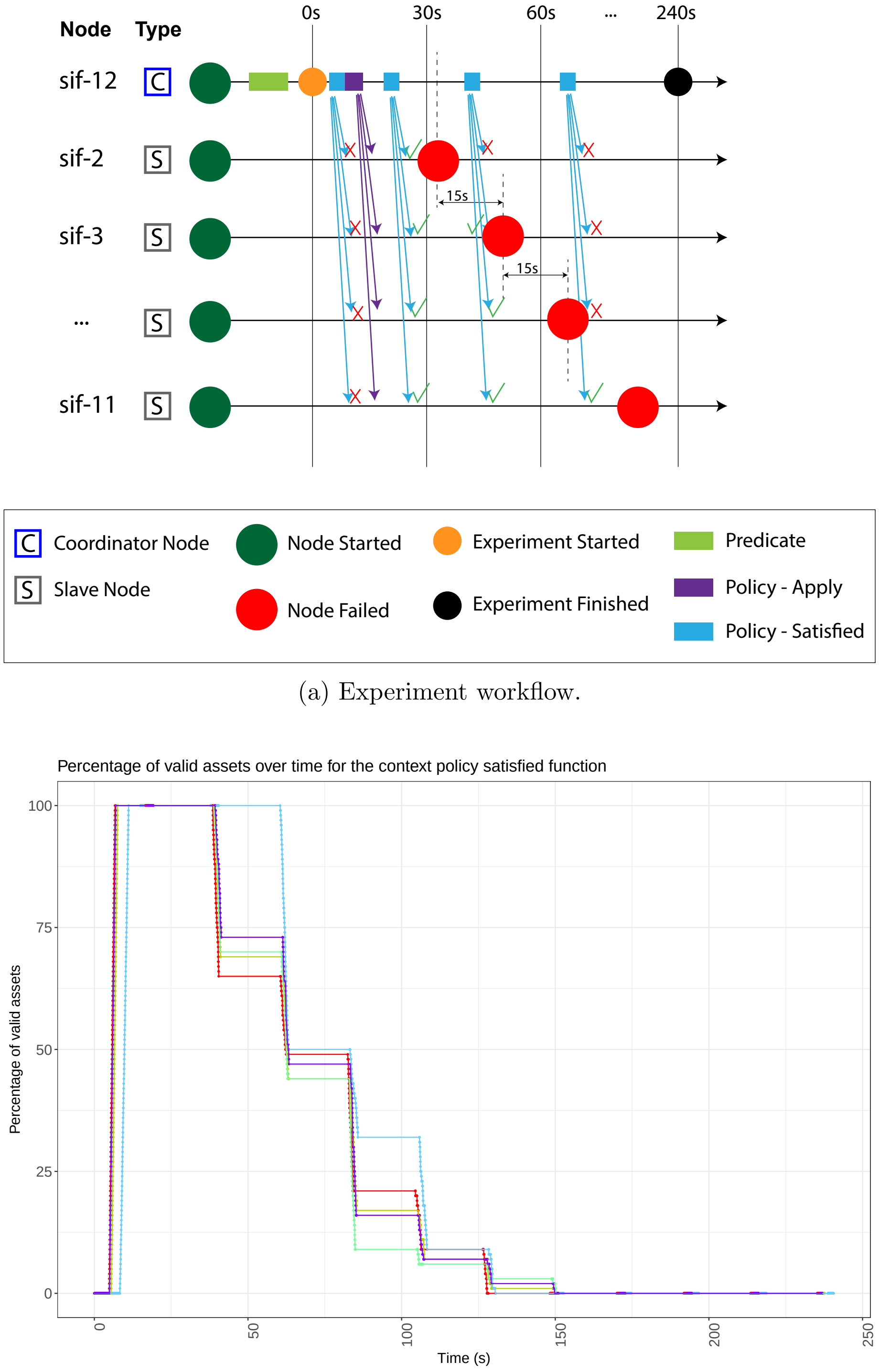
(b) Experiment results.

Figure 8.15: Failure of all the nodes in the codomain: all replica nodes fail.

8.6.5.4 Case 4: Sequential Failure of all the Nodes in the Codomain

This experiment evaluates the ability of a node to detect the unavailability of replicas as multiple nodes of the codomain fail at regular intervals. The experiment is a variation of **Case 3** (see above) and was run over eleven nodes of the BWC, with the slave nodes failing one after the other at intervals of 15 seconds (see Figure 8.16a). The purpose of this experiment is to show that contexts can detect the failure of multiple nodes over time thank to the periodic scheduling of context policy *satisfy function.*

The first part of the experiment is the same as for **Case 3**, with the coordinator replicating the content to the slave nodes and verifying that there are at least three replicas for each asset. After a few seconds, the slave nodes start to fail at regular intervals of 15 seconds, until all nodes are down. As soon as one node is down, all the replicas that it stored become unavailable and the number of valid assets for the policy decreases, as shown in Figure 8.16b. The number of valid assets decreases gradually, since not all slave nodes are down when the next policy *satisfied function* is run. The number of valid assets becomes zero when at least eight slave nodes are down and there are not enough nodes to hold 3 replicas, even for a single asset. The results also show how quickly the coordinator can detect the failure of nodes, which depends on how often the policy *satisfied function* is run. However, in a real world scenario, a node may manage hundreds of thousands or even millions of assets as well as thousands of contexts, thus increasing the frequency by which a context policy is run can be counterproductive.



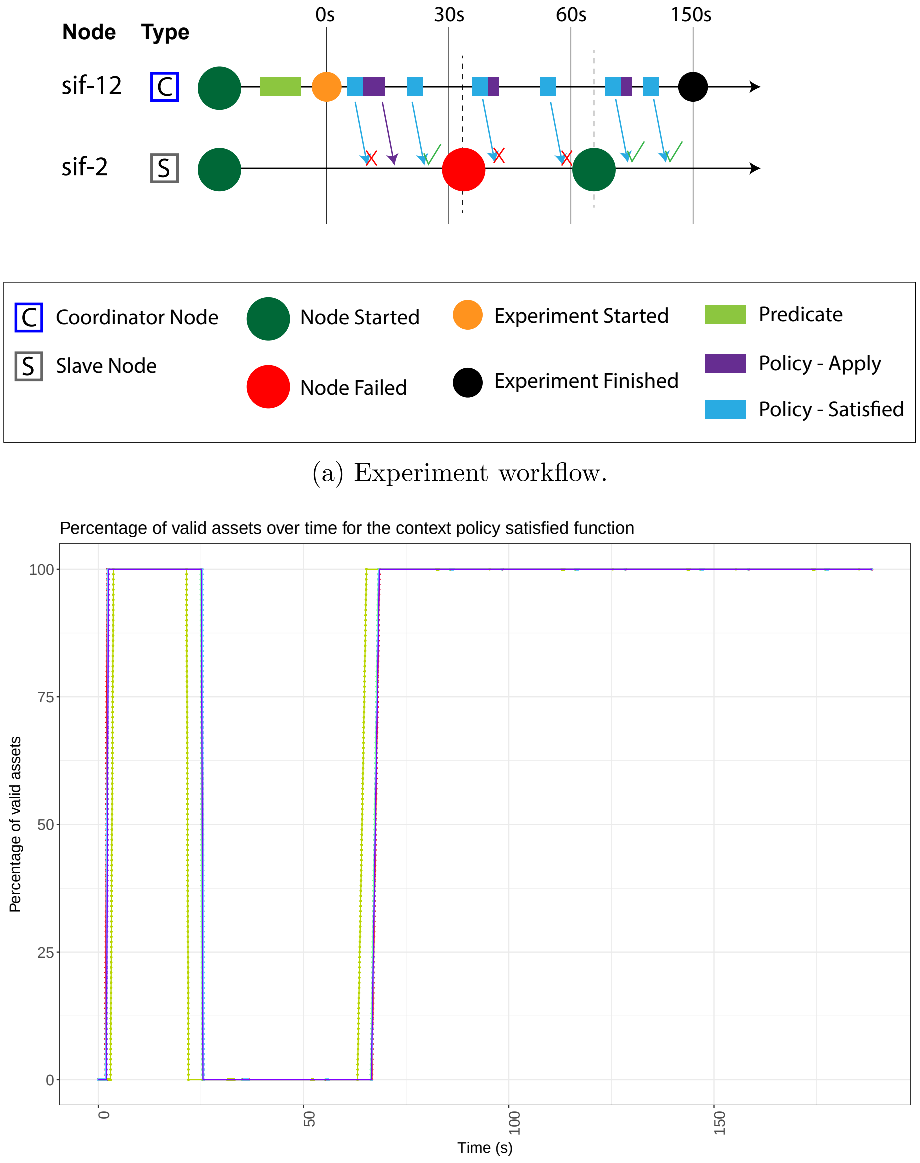
(a) Experiment workflow.

(b) Experiment results.

Figure 8.16: Sequential failure of all the nodes in the codomain: all replica nodes fail one after the other, at 15-second intervals.

8.6.5.5 Case 5: Temporary Failure of a Node

This experiment evaluates the ability of a node to detect the temporary unavailability of replicas. The experiment is run over two nodes of the BWC, a coordinator that replicates content using a context and a slave node that belongs to the context codomain and acts as the storage for the replicas.



(a) Experiment workflow.

(b) Experiment results.

Figure 8.17: Temporary failure of a node: the replica node fails for a short period of time and then comes back alive again. This experiment was run 5 times, even though only two coloured lines are visible.

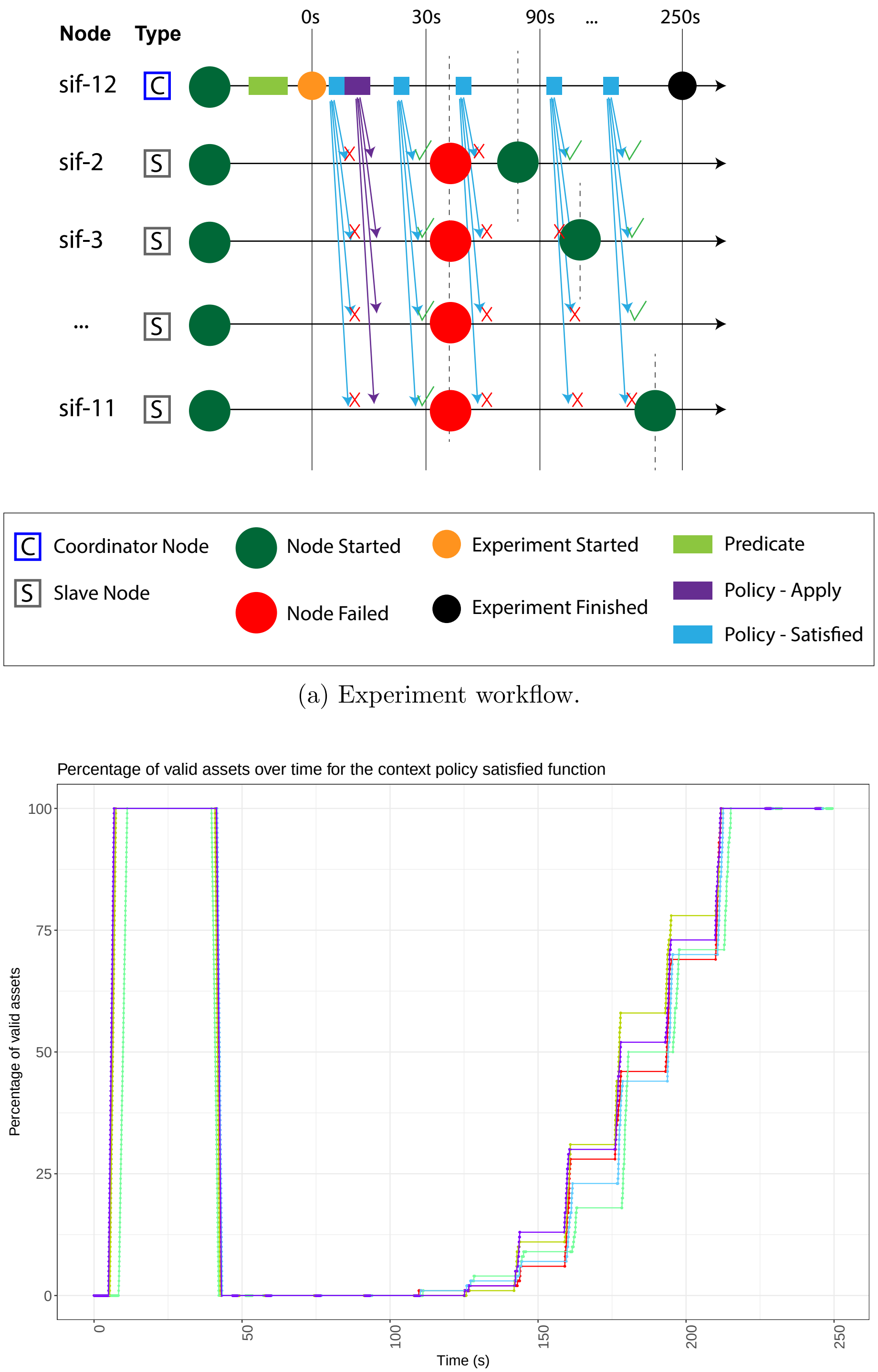
The workflow of this experiment is shown in Figure 8.17a. Similarly to **Case 1**, the coordinator `sif-12` replicates all the managed assets to the slave node `sif-2` in its codomain. Then the slave node fails, event detected by the coordinator when the context policy *satisfied function* is run. However, unlike **Case 1**, in this experiment node `sif-2` becomes available after a few seconds, thus making the replicas it stored available again to the coordinator. The coordinator detects this new change in the state of the codomain only at the next run of the policy *satisfied function* (see Figure 8.17b). Unlike **Case 2**, where the wanted state of the SOS was recovered by replicating the assets to another slave node in the codomain, here the coordinator simply acknowledges that `sif-2` already has the replicated assets and the policy *apply procedure* does not need to perform any more work.

It should be noted that the coordinator can detect the temporary failure of a node if and only if the context *satisfied function* is run within the window of failure. This can be a positive side-effect of the system when having nodes that go down occasionally and for a short amount of time. However, it also means that the system is not very reactive, unless the *satisfied function* is run very often.

8.6.5.6 Case 6: Temporary Failure of all Nodes in the Codomain, which are Restarted Sequentially

This experiment is based on **Case 5** and tests the ability of a node to detect the temporary unavailability of replicas over multiple nodes of the codomain.

The workflow of this experiment is shown in Figure 8.18a. The first part of the experiment is exactly the same as for **Case 3**, but after a while all nodes fail, one of them becomes available again. The other slave nodes become available too, one after the other at intervals of 15 seconds each, until eventually all slave nodes are available. The context used for this experiment replicates the 100 assets of the *R_100KBx100* dataset across the nodes in the codomain with a replication factor of 3. As soon as at least three nodes in the codomain become available, the context validates the assets for which there are at least three replicas, as shown in Figure 8.18b. The number of valid assets increases gradually, in steps. It should also be noted that the step increases become larger as more nodes become available, as a sign that there are more nodes where the replicas are stored.



(a) Experiment workflow.

(b) Experiment results.

Figure 8.18: Temporary failure of all nodes in the codomain, which are restarted sequentially: all the replica nodes fail, but then come back alive again one after the other, at 15-second intervals.

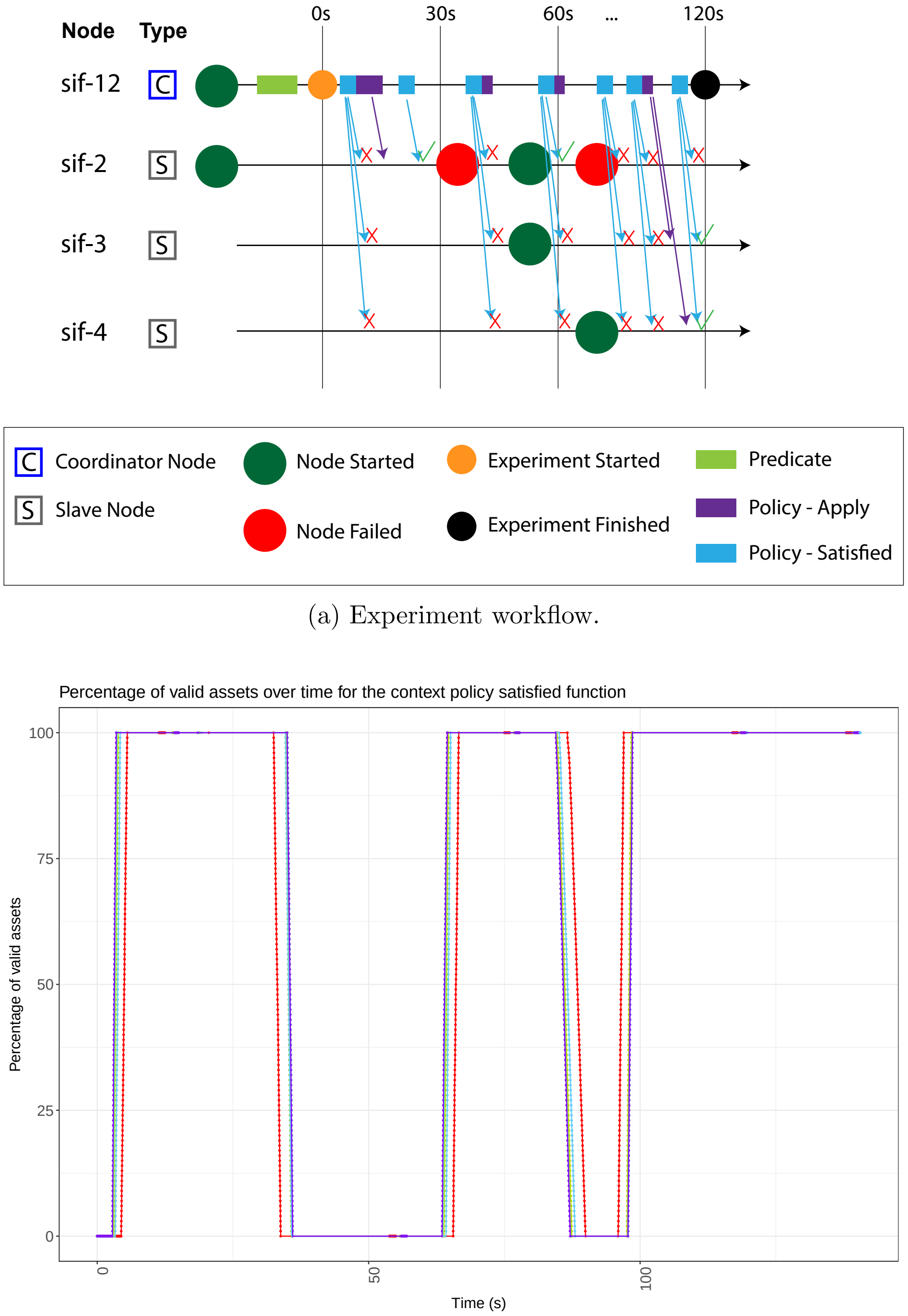
The conclusion drawn from this experiment is that a context can recover reasonably quickly from the failure of multiple nodes in the codomain, especially thanks to the policy *satisfied function* which helps avoiding unnecessary data replications. In this particular case, the policy *apply procedure* was not run. Nonetheless, if the coordinator were to run the *apply procedure*, the codomain could have ended up storing more assets replicas than specified by the policy replication factor. This scenario is observed in **Case 7** and can be problematic when working with a large number of assets. As future work, one might devise a mechanism to restrict the actions of policies within certain boundaries to overcome this issue.

8.6.5.7 Case 7: Change in the Number of Nodes in the Codomain, while some Fail.

This experiment tests the ability of a node to react to multiple changes in the codomain, in particular as the number of nodes changes over time. The workflow of the experiment is shown in Figure 8.19a.

The context used in the experiment has a codomain of four nodes — the slave nodes plus the coordinator — and its policy replicates the assets from the *R_100KBx100* dataset with a replication factor of two (*i.e.*, one copy stored locally and one remotely). To start with, only one of the slave nodes is available, so all the replicas are stored in `sif-2` at the beginning. After a few seconds `sif-2` becomes unavailable and the policy *satisfied function* returns false for all the assets, as shown in Figure 8.19b. Soon after, both slave nodes `sif-2` and `sif-3` become available, however, since the former already has the replicas from before, no active replication of the assets is involved. After a few more seconds, `sif-2` fails again, while a third slave node, `sif-4`, becomes available. As a result, the replicas are now stored in both `sif-3` and `sif-4`.

This experiment confirms that the model to manage policies used by the context model works reasonably well when the state of the codomain changes. In scenarios such as the one just presented, however, the context might replicate more content than needed.



(a) Experiment workflow.

(b) Experiment results.

Figure 8.19: Change in the number of nodes in the codomain, while some fail: all or some of the nodes in the codomain fail, while the number of the nodes in the codomain changes too.

8.6.5.8 Summary of Contexts under Failure

This section has illustrated the behaviour of contexts and their policies when nodes in the codomain fail. The behaviours observed outline the following properties regarding contexts and their policies:

- Policy *apply procedures* can change the state of the SOS.
- Policy *satisfy functions* are generally not expensive to execute, as shown in Section 8.6.3, and can be run multiple times to detect changes in the state of the SOS.
- Contexts that have codomains of multiple nodes are more resilient to failures, as shown in **Cases 2** and **7**.
- In the face of partial faults in the codomain, contexts can run policy *apply procedures* as a repair mechanism to re-establish the wanted state of the SOS. This has a cost which was observed in the experiments in Sections 8.6.3 and 8.6.4.

These seven scenarios, and the results presented, should be considered as illustrative examples of what contexts are capable of, rather than as a set of all the possible use cases that contexts can handle. Contexts where domains consist of more than one node should also be explored when considering failure. The current experimental framework, however, does not allow multiple coordinators to be tracked and monitored. This and other scenarios, which could derive from combinations of different types of predicates, policies, domains, and codomains, are worth exploring as part of future work.

8.7 Conclusions

This chapter has presented two classes of experiments: one that benchmarks the basic operations of the reference implementation and one that evaluates the performance and behaviour of the different components of the SOS context model.

The results of the benchmarks show that the SOS reference implementation performs reasonably well for reading data, but less so when writing it. The hashing calculations represent a bottleneck, which could be removed by pipelining the data hashing and writing operations. Performing expensive operations in parallel, such as data replication, can also improve the overall performance of a SOS node.

The results of the context related experiments demonstrate that the context model is a feasible solution for managing content in a small distributed system. The experiments also show the behaviour of multiple SOS nodes interacting through the context model. The results presented should be used as guidelines for designing new contexts and as a starting point for future designs and experiments regarding the context model.

Conclusions and Future Work

Data is at the centre of our digital life and managing it has become challenging, both from the user and system perspective. This thesis provides a thorough overview of data storage systems and the design of a model to manage shared mutable data in a distributed environment. Three challenges, in particular, are addressed: (1) how to construct abstractions such that data is accessible irrespective of its location and versionable in a large distributed system; (2) how to provide transparent and secure data management irrespective of locations, users, applications, and services; and (3) how to allow users to protect and control content through well-defined rules in a distributed system.

This concluding chapter briefly summarises this work and outlines its main research contributions. Then it discusses the limitations of the evaluation presented and presents some directions for future work.

9.1 Thesis Summary

This thesis began with an *Introduction* Chapter, which outlines the problems and challenges in distributed storage systems and proposes two research hypotheses, which define the scope of the work discussed in the chapters following.

Chapter 2 outlines the basic data management concepts relevant to this work and an overview of data management systems (DMS). Chapter 3 continues by presenting the state of the art DMS and how they address the challenges identified in the first chapter, with a particular focus on distributed DMS. The requirements of a distributed DMS — based on the *Background* and *Literature Review* Chapters and in relation to the hypotheses presented in the *Introduction* Chapter — are listed in Chapter 4, in terms of end-users, model, and architecture requirements. These define the basis for the design of the *Sea of Stuff*, a model to manage shared mutable data in a distributed environment, presented in Chapter 5.

The *Sea of Stuff* model and architecture is based on immutable, self-describing, recomputable, and content-addressable entities represented by structured manifests. The *Sea of Stuff* allows data to be managed and accessed irrespective of its location, aggregated by reference and de-duplicated, and versioned. It also uses a user-role model to digitally sign and protect data, metadata, and any other entity of its model. Automatic content aggregation and control over multiple nodes is achieved via the context model. A reference implementation of the *Sea of Stuff* has been built and described in Chapter 6.

Finally, Chapter 7 compares the SOS design and implementation qualitatively against the relevant DMS presented in the literature review, while Chapter 8 evaluates some aspects of the *Sea of Stuff* experimentally, with a particular focus on contexts and their behaviour in a small distributed system. These evaluations suggest that the *Sea of Stuff* is a viable DMS, but more development on the prototype and additional experiments need to be run to conclusively affirm such a statement.

9.2 Review of Contributions

The main contributions of this thesis are a comprehensive literature review on data management and the design and implementation of the *Sea of Stuff*, a model for shared mutable data in a distributed system.

Chapters 2 and 3 provide an understanding of the basic properties of distributed storage systems and an overview of such systems. The data abstractions, access control models, security mechanisms, metadata management, and architecture are described and discussed for each storage system taken into account.

The contribution provided by Chapter 5 is the design of the *Sea of Stuff* model and architecture. The key aspects that distinguish the SOS design from other work are the following:

- Content is versionable, so that each version can be distributed independently of the asset it belongs to.
- Metadata is extrinsic to the data it describes and it can be distributed and accessed independently of where its associated content is.
- A multi-role model allows content to be digitally signed and/or protected at different granularities.
- End-users can define a set of rules to identify and control content over a distributed set of heterogeneous machines.
- The *Sea of Stuff* model is independent of the reference implementation presented in this work and the underlying file system. Storage systems supporting the *Sea of Stuff* model can interact consistently with each other while preserving data and its associated metadata.

A prototype of the *Sea of Stuff* is described in Chapter 6 and made available at `https://github.com/sea-of-stuff` under the GPL-3.0 license.

The *Sea of Stuff* model and architecture designs are compared qualitatively in Chapter 7. This evaluation shows how the *Sea of Stuff* compares to other related systems. Chapter 8 evaluates the viability of the context model, as provided by the *Sea of Stuff* prototype, within a small distributed environment.

9.3 Limitations of Evaluations

The evaluations presented in this thesis suggest that the *Sea of Stuff* can be a viable distributed storage data model. However, there are also a number of elements that have not been evaluated that limit the value and scope of these evaluations.

Chapter 7 compares the models and architectures of various storage systems against the *Sea of Stuff*, but it does not provide an understanding of how these solutions perform under similar circumstances. For example, it is not clear how the *Sea of Stuff* scales when managing millions, or even billions, of entities over hundreds of thousands of nodes and how it would compare with other systems, like OceanStore and IPFS. Moreover, the comparison for some of the systems is limited by the knowledge we have of closed source systems.

Regarding the evaluation reported in Chapter 8, none of the experiments presented has been run on a heterogeneous distributed environment that could resemble a real world scenario. Thus, it is not possible to assert with confidence what the behaviour of the *Sea of Stuff* is when used by real users. Similarly, all the experiments were run against synthetic datasets, rather than data collections resembling the digital life of real end-users.

The evaluation of the *Sea of Stuff* context model explores only a few aspects of contexts and their components. Firstly, the predicates and policies evaluated represent only a very small subset of what they could be used for. Secondly, contexts have been tested against domains and codomains of homogeneous nodes, unlike the heterogeneous real world. Finally, while distributed systems can face a large variety of failures, only node failures within the codomain were evaluated. A more extensive evaluation of the context model needs to be performed to understand its viability in a real-world scenario.

9.4 Future Work

The evaluation chapters have demonstrated the viability of the *Sea of Stuff* and how it satisfied the requirements outlined in Chapter 4. The *Sea of Stuff* model and its implementation, however, are far from being good enough that they can be deployed in the real-world. This section discusses possible future directions for the work presented in this thesis and this research topic in general.

9.4.1 The Sea of Stuff Model and Architecture

- Additional research needs to be carried regarding the management of dynamic content (*i.e.*, content that changes frequently).

- In the current design, the asset entity is used to version atoms and compounds, but it could potentially be used to version also other types of entities, such as roles.

- The design proposed in this thesis does not define any particular strategy about where to store content and where to retrieve it from. Future research should investigate the use of DHT as well as reputation models based on nodes/users/roles.

- Metadata is an important element of the *Sea of Stuff* model, since it describes assets and provides useful information for building contexts. The metadata entity could be integrated with the compound, so that it would be possible to have better deduplication, useful when handling large amount of metadata. The metadata model could also be integrated with the Resource Description Framework [89] to enable richer relationships to be defined.

- The *Sea of Stuff* uses the context model to manage content in an automatic manner within a distributed system. Further research on contexts should explore the design of predicates that could return different types of values (*e.g.*, integer/real numbers, set of values, object values), other than boolean values only. Exploring better and richer semantics for policies is also essential to understand the actual extent of what policies can be used for and how policy conflicts can be resolved. Potential solutions could involve the use of a domain-specific language to express and limit computation and/or building a priority distributed data structure to determine what policy to accept or reject for a particular asset. Finally, researchers could explore the ability to combine contexts together under the AND/OR boolean operators to express richer rules starting from pre-existing contexts.

9.4.2 Security

Security is a very important element of any distributed storage system. Further versions of the SOS should provide at-rest and in-transit protection of atoms and manifests by default. Researchers should also explore any potential security risks related to contexts and what countermeasures can be taken.

9.4.3 Sea of Stuff Integration with Existing Technology

The *Sea of Stuff* is a generic data model based on concepts and mechanisms used by many other DMS. Exploring how the *Sea of Stuff* model can be integrated with existing technology, from cloud to P2P storage solutions, is a valuable direction for future work. Understanding the integration between the *Sea of Stuff* and already-existing blockchain technology is also particularly relevant, as shown by the notable amount of interest the community has on projects like SIA [225], StorJ [226] and Filecoin [227]. Moreover, integrating the fundamentals of smart contracts [228] with the *Sea of Stuff* context model could open-up new possibilities in terms of what contexts are and how users interact with them.

9.5 Concluding Remarks

In summary, this thesis gives an in-depth overview of data management solutions and presents and evaluates the design of a distributed data model that provides: location abstraction, rich metadata management, data protection, versioning, autonomic data management, and resiliency.

The first part of this thesis contributes to a better understanding of the existing data management solutions, how they compare, and what their advantages and disadvantages are. It is then shown that a data model based on immutable, self-describing, re-computable, and content-addressable entities can be used to build a distributed data management system that allows data to be managed at scale, while providing the properties listed above. With the *Sea of Stuff*, not only end-users are not bound to devices, services, and applications any longer, but they can also design and implement rules that allows data to be organised and controlled in better and smarter ways.

The comparative and experimental evaluations presented in this work suggest that the *Sea of Stuff* is a feasible model and architecture, but additional work, as suggested in this chapter, is needed to conclusively affirm its viability in the real-world.

Bibliography

[1] Y. N. Harari, *Sapiens: A Brief History of Humankind.* Random House, 2014.

[2] N. Wirth, *Algorithms + Data Structures = Programs.* Prentice Hall PTR, 1978.

[3] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young, "Report of the ACM Task Force on the Core of Computer Science," *Communications of the ACM*, 1988.

[4] The Linux Information Project. Computer Science Definition. Last accessed on 14/04/2018. [Online]. Available: http://linfo.org/computerscience.html

[5] P. J. Denning, "Computing is a Natural Science," *Communications of the ACM*, vol. 50, no. 7, pp. 13–18, 2007.

[6] M. Tedre, *The Science of Computing: Shaping a Discipline.* CRC Press, 2014.

[7] J. Gantz and D. Reinsel, "The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East," 2012.

[8] D. Reinsel, J. Gantz, and J. Rydning, "Data Age 2025: The Evolution of Data to Life-Critical," *Technical Report, International Data Corporation*, Last accessed on 15/01/2018. [Online]. Available: https://seagate.com/files/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf

[9] M. I. Seltzer and N. Murphy, "Hierarchical File Systems Are Dead," *Usenix HotOS*, 2009.

[10] J. W. O'Toole and D. K. Gifford, "Names should mean What, not Where," *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systems structuring*, pp. 1–5, 1992.

[11] S. Nickell, "A Cognitive Defense of Associative Interfaces for Object Reference," 2006, Last accessed on 23/01/2018. [Online]. Available: http://gnome.org/~seth/storage/associative-interfaces.pdf

[12] J. Nielsen, "The Impending Demise of the File System," *IEEE Software*, vol. 13, no. 2, pp. 100–101, 1996.

[13] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber, "Scientific Data Management in the Coming Decade," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 34–41, 2005.

[14] R. Harper, S. Lindley, E. Thereska, R. Banks, P. Gosset, G. Smyth, W. Odom, and E. Whitworth, "What is a File?" *Proceedings of the 2013 conference on Computer supported cooperative work*, pp. 1125–1136, 2013.

[15] J. Martin, *Managing the Data Base Environment.* Prentice Hall PTR, 1983.

[16] F. Halasz and T. P. Moran, "Analogy Considered Harmful," *Proceedings of the 1982 Conference on Human Factors in Computing Systems*, pp. 383–386, 1982.

[17] S. E. Lindley, C. C. Marshall, R. Banks, A. Sellen, and T. Regan, "Rethinking the Web as a Personal Archive," *Proceedings of the 22nd International Conference on World Wide Web*, pp. 749–760, 2013.

[18] Dropbox, Inc. Dropbox. Last accessed on 20/10/2017. [Online]. Available: https://dropbox.com

[19] Microsoft, Corp. Microsoft OneDrive. Last accessed on 20/10/2017. [Online]. Available: https://onedrive.live.com/

[20] Amazon.com, Inc. Amazon Simple Storage Service. Last accessed on 20/10/2017. [Online]. Available: https://aws.amazon.com/s3

[21] Rackspace, Inc. Rackspace - Scalable Cloud Object Storage. Last accessed on 20/10/2017. [Online]. Available: https://rackspace.com/cloud/files

[22] R. Buyya, J. Broberg, and A. M. Goscinski, *Cloud Computing: Principles and Paradigms*. John Wiley & Sons, 2010, vol. 87.

[23] K. M. Khan and Q. Malluhi, "Establishing Trust in Cloud Computing," *IT professional*, vol. 12, no. 5, pp. 20–27, 2010.

[24] B. H. Kim, W. Huang, and D. Lie, "Unity: Secure and Durable Personal Cloud Storage," *Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop*, pp. 31–36, 2012.

[25] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud Storage with Minimal Trust," *ACM Transactions on Computer Systems*, vol. 29, no. 4, p. 12, 2011.

[26] D. Slamanig and C. Hanser, "On Cloud Storage and the Cloud of Clouds Approach," *International Conference for Internet Technology And Secured Transactions*, pp. 649–655, 2012.

[27] J. Shen, J. Gu, Y. Zhou, and X. Wang, "Cloud-of-Clouds Storage Made Efficient: A Pipeline-Based Approach," *IEEE International Conference on Web Services*, pp. 724–727, 2016.

[28] H.-S. Yeo, X.-S. Phang, H.-J. Lee, and H. Lim, "Leveraging Client-Side Storage Techniques for Enhanced use of Multiple Consumer Cloud Storage Services on Resource-Constrained Mobile Devices," *Journal of Network and Computer Applications*, vol. 43, pp. 142–156, 2014.

[29] J. L. Gonzalez, J. C. Perez, V. J. Sosa-Sosa, L. M. Sanchez, and B. Bergua, "SkyCDS: A Resilient Content Delivery Service based on Diversified Cloud Storage," *Simulation Modelling Practice and Theory*, vol. 54, pp. 64–85, 2015.

[30] B. Di Martino, D. Petcu, R. Cossu, P. Goncalves, T. Máhr, and M. Loichate, "Building a Mosaic of Clouds," *European Conference on Parallel Processing*, pp. 571–578, 2010.

[31] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed. Addison-wesley Reading, 2010.

[32] A. S. Tanenbaum and M. Van Steen, *Distributed Systems: Principles and Paradigms*, 2nd ed. Prentice-Hall, 2007.

[33] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax," Internet Requests for Comments, RFC Editor, RFC 2396, August 1998, Last accessed on 20/07/2018. [Online]. Available: https://rfc-editor.org/rfc/rfc2396.txt

[34] Tim Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," Internet Requests for Comments, RFC Editor, RFC 3986, January 2005, Last accessed on 20/07/2018. [Online]. Available: https://rfc-editor.org/rfc/rfc3986.txt

[35] P. Leach, M. Mealling, and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace," Internet Requests for Comments, RFC Editor, RFC 4122, July 2005, Last accessed on 20/07/2018. [Online]. Available: https://rfc-editor.org/rfc/rfc4122.txt

[36] J. Greenberg, "Metadata and the World Wide Web," *Encyclopedia of library and information science*, vol. 3, pp. 1876–1888, 2003.

[37] Press, NISO, "Understanding Metadata," *National Information Standards*, vol. 20, 2004.

[38] J. Riley, "Understanding Metadata: What is Metadata, and What is it For?: A Primer," 2017.

[39] E. A. Brewer, "Towards Robust Distributed Systems," *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, vol. 7, 2000.

[40] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," *ACM Sigact News*, vol. 33, no. 2, pp. 51–59, 2002.

[41] E. Brewer, "CAP Twelve Years Later: How the 'Rules' have Changed," *IEEE Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[42] D. Abadi, "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story," *IEEE Computer*, vol. 45, no. 2, pp. 37–42, 2012.

[43] A. Ailamaki, V. Kantere, and D. Dash, "Managing Scientific Data," *Communications of the ACM*, vol. 53, no. 6, pp. 68–78, 2010.

[44] J. N. Gray, "Notes on Data Base Operating Systems." Springer, 1978, pp. 393–481.

[45] L. Lamport, "The Part-Time Parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[46] I. S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[47] A. B. Primer, "Erasure Codes for Storage Systems," *Usenix ;login:*, vol. 38, no. 6, pp. 44–50, 2013.

[48] H. Weatherspoon and J. D. Kubiatowicz, "Erasure Coding vs. Replication: A Quantitative Comparison," *International Workshop on Peer-to-Peer Systems*, pp. 328–337, 2002.

[49] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman, "Efficient Erasure Correcting Codes," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 569–584, 2001.

[50] D. A. Patterson, G. Gibson, and R. H. Katz, *A Case for Redundant Arrays of Inexpensive Disks (RAID).* ACM, 1988, vol. 17, no. 3.

[51] H.-Y. Lin and W.-G. Tzeng, "A Secure Erasure Code-based Cloud Storage System with Secure Data Forwarding," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 6, pp. 995–1003, 2012.

[52] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh, "A Survey on Network Codes for Distributed Storage," *Proceedings of the IEEE*, vol. 99, no. 3, pp. 476–489, 2011.

[53] D. Gligoroski, K. Kralevska, R. E. Jensen, and P. Simonsen, "Network Traffic Driven Storage Repair," *arXiv preprint arXiv:1803.03682*, 2018.

[54] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing Elephants: Novel Erasure Codes for Big Data," *Proceedings of the VLDB Endowment*, 2013.

[55] S. Caron, F. Giroire, D. Mazauric, J. Monteiro, and S. Pérennes, "P2P Storage Systems: Study of Different Placement Policies," *Peer-to-Peer Networking and Applications*, vol. 7, no. 4, pp. 427–443, 2014.

[56] Z. Wilcox-O'Hearn and B. Warner, "Tahoe: the Least-Authority Filesystem," *Proceedings of the 4th ACM International Workshop on Storage Security and Survivability*, pp. 21–26, 2008.

[57] D. Irvine, "MaidSAFE Distributed File System," *Technical Report, MaidSafe.net*, 2010, Last accessed on 15/03/2018. [Online]. Available: http://docs.maidsafe.net/Whitepapers/pdf/MaidSafeDistributedFileSystem.pdf

[58] A. Grünbacher, "POSIX Access Control Lists on Linux." *Usenix Annual Technical Conference, FREENIX Track*, pp. 259–272, 2003.

[59] Microsoft Corp. Access Control Lists. Last accessed on 04/02/2018. [Online]. Available: https://msdn.microsoft.com/library/windows/desktop/aa374872

[60] R. S. Fabry, "Capability-based Addressing," *Communications of the ACM*, vol. 17, no. 7, pp. 403–412, 1974.

[61] J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[62] B. W. Lampson, "Dynamic Protection Structures," *Proceedings of the AFIPS Fall Joint Computer Conference*, pp. 27–38, 1969.

[63] B. W. Lampson, "Protection," *ACM SIGOPS Operating Systems Review*, vol. 8, no. 1, pp. 18–24, 1974.

[64] R. Rivest, "The MD5 Message-Digest Algorithm," Internet Requests for Comments, RFC Editor, RFC 1321, April 1992, Last accessed on 20/07/2018. [Online]. Available: https://rfc-editor.org/rfc/rfc1321.txt

[65] H. Dobbertin, "The Status of MD5 After a Recent Attack," *CryptoBytes*, vol. 2, no. 2, 1996.

[66] X. Wang and H. Yu, "How to Break MD5 and Other Hash Functions," *Eurocrypt*, vol. 3494, pp. 19–35, 2005.

[67] E. Biham and R. Chen, "Near-collisions of SHA-0," *Annual International Cryptology Conference*, pp. 290–305, 2004.

[68] X. Wang, H. Yu, and Y. L. Yin, "Efficient Collision Search Attacks on SHA-0," *Annual International Cryptology Conference*, pp. 1–16, 2005.

[69] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The First Collision for Full SHA-1," *IACR Cryptology ePrint Archive*, vol. 2017, p. 190, 2017.

[70] W. Diffie, "The First Ten Years of Public-Key Cryptography," *Proceedings of the IEEE*, vol. 76, no. 5, pp. 560–577, 1988.

[71] S. Landau, “Standing the Test of Time: The Data Encryption Standard,” *Notices of AMS*, vol. 47, no. 3, pp. 341–349, 2000.

[72] R. L. Rivest, A. Shamir, and L. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[73] R. C. Merkle, “A Certified Digital Signature,” *Conference on the Theory and Application of Cryptology*, pp. 218–238, 1989.

[74] Y. Zheng, “Digital Signcryption or How to Achieve Cost(Signature & Encryption)≪ Cost(Signature) + Cost(Encryption),” *Advances in Cryptology—Crypto ’97*, pp. 165–179, 1997.

[75] P. Gallagher, “Digital Signature Standard (DSS),” *Federal Information Processing Standards Publications*, pp. 186–3, 2013.

[76] T. Rizzo, “WinFS 101: Introducing the New Windows File System,” *Longhorn Developer Center Home: Headline Archive: WinFS*, vol. 101, pp. 1–5, 2004, Last accessed on 23/05/2018. [Online]. Available: https://msdn.microsoft.com/library/aa480687.aspx

[77] J. S. Heidemann, “Stackable Design of File Systems,” no. CSD-950032, pp. xvi + 105, Sep. 1995, Last accessed on 10/02/2018. [Online]. Available: http://isi.edu/~johnh/PAPERS/Heidemann95e.html

[78] S. R. Kleiman, “Vnodes: An Architecture for Multiple File System Types in Sun UNIX,” *Proceedings of the Usenix Summer Technical Conference*, vol. 86, pp. 238–247, 1986.

[79] H. Custer, “Inside the Windows NT File System,” *Microsoft Press Redmond, Washington*, 1994.

[80] Microsoft Corp. Description of the FAT32 File System. Last accessed on 29/11/2017. [Online]. Available: https://support.microsoft.com/en-gb/help/154997/description-of-the-fat32-file-system

[81] Microsoft Corp. Resilient File System (ReFS) Overview. Last accessed on 29/11/2017. [Online]. Available: http://docs.microsoft.com/en-US/windows-server/storage/refs/refs-overview

[82] Microsoft Corp. Hard Links and Junctions. Last accessed on 04/02/2018. [Online]. Available: https://msdn.microsoft.com/library/windows/desktop/aa365006

[83] Wikimedia Commons - User:Sven. FUSE Structure. Last accessed on 27/03/2018. [Online]. Available: https://commons.wikimedia.org/wiki/File:FUSE_structure.svg

[84] The Linux Information Project. Database Definition. Last accessed on 10/02/2018. [Online]. Available: http://linfo.org/database.html

[85] G. S. Fowler, "cql – A Flat File Database Query Language," *Usenix Winter*, pp. 11–21, 1994.

[86] A. Dearle, R. Di Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, F. Vaughan *et al.*, "Grasshopper: An Orthogonally Persistent Operating System," *Computing Systems*, vol. 7, no. 3, pp. 289–312, 1994.

[87] M. Atkinson and R. Morrison, "Orthogonally Persistent Object Systems," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. 4, no. 3, pp. 319–402, 1995.

[88] M. P. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence, "An Orthogonally Persistent Java," *ACM Sigmod Record*, vol. 25, no. 4, pp. 68–75, 1996.

[89] O. Lassila and R. R. Swick, "Resource Description Framework (RDF) Model and Syntax Specification," 1999, Last accessed on 20/06/2018. [Online]. Available: https://w3.org/TR/1999/REC-rdf-syntax-19990222/

[90] The Apache Software Foundation. Apache Subversion. Last accessed on 23/04/2018. [Online]. Available: https://subversion.apache.org/

[91] W. F. Tichy, "RCS: a System for Version Control," *Software: Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985.

[92] L. Torvalds and J. Hamano, "Git: Fast Version Control System," 2010, Last accessed on 02/02/2018. [Online]. Available: http://git-scm.com

[93] M. Mackall, "Towards a Better SCM: Revlog and Mercurial," *Proceedings of the Ottawa Linux Symposium*, vol. 2, pp. 83–90, 2006.

[94] M. Satyanarayanan, "A Survey of Distributed File Systems," *Annual Review of Computer Science*, 1989.

[95] M. Placek and R. Buyya, "A Taxonomy of Distributed Storage Systems," *Technical Report, Computing and Distributed Systems Department, University of Melbourne*, 2006.

[96] T. D. Thanh, S. Mohan, E. Choi, S. Kim, and P. Kim, "A Taxonomy and Survey on Distributed File Systems," *Fourth International Conference on Networked Computing and Advanced Information Management*, vol. 1, pp. 144–149, 2008.

[97] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem," *Proceedings of the Usenix Summer Technical Conference*, pp. 119–130, 1985.

[98] J. H. Howard, *An Overview of the Andrew File System.* Carnegie Mellon University, Information Technology Center, 1988.

[99] Microsoft Corp. (2013, Jun.) Server Message Block Overview. Last accessed on 22/02/2018. [Online]. Available: https://technet.microsoft.com/en-us/library/hh831795.aspx

[100] P. J. Leach and D. Naik, “A Common Internet File System (CIFS/1.0) Protocol,” *Internet-Draft, IETF*, 1997, Last accessed on 02/02/2018. [Online]. Available: https://tools.ietf.org/html/draft-leach-cifs-v1-spec-01

[101] B. Callaghan, B. Pawlowski, and P. Staubach, “NFS Version 3 Protocol Specification,” Internet Requests for Comments, RFC Editor, RFC 1813, June 1995, Last accessed on 20/07/2018. [Online]. Available: https://rfc-editor.org/rfc/rfc1813.txt

[102] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, “NFS Version 4 Protocol,” Internet Requests for Comments, RFC Editor, RFC 3010, December 2000, Last accessed on 20/07/2018. [Online]. Available: https://rfc-editor.org/rfc/rfc3010.txt

[103] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, “Network File System (NFS) Version 4 Protocol,” Internet Requests for Comments, RFC Editor, RFC 3530, April 2003, Last accessed on 20/07/2018. [Online]. Available: https://rfc-editor.org/rfc/rfc3530.txt

[104] T. Haynes and D. Noveck, “Network File System (NFS) Version 4 Protocol,” Internet Requests for Comments, RFC Editor, RFC 7530, March 2015, Last accessed on 20/07/2018. [Online]. Available: https://rfc-editor.org/rfc/rfc7530.txt

[105] Trend Micro, Inc. Installing OfficeScan 10.6 on Windows 2008 R2 cluster environment. Last accessed on 27/03/2018. [Online]. Available: https://success.trendmicro.com/solution/1059586-installing-officescan-10-6-on-windows-2008-r2-cluster-environment

[106] Seagate Technology, LLC. Lustre. Last accessed on 28/11/2017. [Online]. Available: http://lustre.org

[107] Oracle Corp. Project: OCFS2. Last accessed on 28/11/2017. [Online]. Available: https://oss.oracle.com/projects/ocfs2/

[108] Red Hat, Inc. Global File System 2. Last accessed on 28/11/2017. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html-single/global_file_system_2/index

[109] M. Singhal and N. G. Shivaratri, *Advanced Concepts in Operating Systems.* McGraw-Hill, Inc., 1994.

[110] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," *IEEE 26th Symposium on Mass Storage Systems and Technologies*, pp. 1–10, 2010.

[111] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.

[112] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "OceanStore: An architecture for Global-Scale Persistent Storage," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 190–201, 2000.

[113] J. Benet, "IPFS – Content Addressed, Versioned, P2P File System," *arXiv preprint arXiv:1407.3561*, 2014.

[114] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009.

[115] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A Break in the Clouds: Towards a Cloud Definition," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 1, pp. 50–55, 2008.

[116] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," *Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg*, 2011.

[117] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," *Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley*, 2009.

[118] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A View of Cloud Computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[119] Amazon.com, Inc. Amazon DynamoDB - NoSQL Cloud Database Service. Last accessed on 28/11/2017. [Online]. Available: https://aws.amazon.com/dynamodb

[120] Digital Ocean, Inc. Spaces on Digital Ocean. Last accessed on 10/04/2018. [Online]. Available: https://digitalocean.com/products/spaces/

[121] Amazon.com, Inc. AWS Elastic Beanstalk. Last accessed on 28/11/2017. [Online]. Available: https://aws.amazon.com/elasticbeanstalk/

[122] Heroku.com. Heroku. Last accessed on 28/03/2018. [Online]. Available: https://heroku.com

[123] Google, LLC. App Engine – Google Cloud Platform. Last accessed on 28/11/2017. [Online]. Available: https://cloud.google.com/appengine

[124] Google, LLC. Google Drive – Cloud Storage & File Backup for Photos, Docs & More. Last accessed on 28/11/2017. [Online]. Available: https://google.com/drive

[125] Backblaze, Inc. Backblaze B2 - Overview. Last accessed on 10/04/2018. [Online]. Available: https://backblaze.com/b2/docs/

[126] M. Mesnier, G. R. Ganger, and E. Riedel, "Object-Based Storage," *IEEE Communications Magazine*, vol. 41, no. 8, pp. 84–90, 2003.

[127] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, "Object Storage: The Future Building Block for Storage Systems," *Local to Global Data Interoperability-Challenges and Technologies, 2005*, pp. 119–123, 2005.

[128] J. C. Mogul, "Representing Information about Files," Ph.D. dissertation, Stanford University, 1986.

[129] Macintosh Inside, *More Macintosh Toolbox.* Addison Wesley, 1993.

[130] D. Giampaolo, *Practical File System Design.* Morgan Kaufmann Publishers, 1998.

[131] Haiku, Inc. Haiku Project. Last accessed on 22/04/2018. [Online]. Available: https://haiku-os.org/

[132] S. Bloehdorn and M. Völkel, "TagFS – Tag Semantics for Hierarchical File Systems," *Proceedings of the 6th International Conference on Knowledge Management*, vol. 8, 2006.

[133] Apple, Inc. macOS Sierra: Use Tags to Organize Files. Last accessed on 22/04/2018. [Online]. Available: https://support.apple.com/kb/PH25325?locale=en_US

[134] G. Klyne and J. J. Carroll, "Resource Description Framework (RDF): Concepts and Abstract Syntax," 2006, Last accessed on 20/06/2018. [Online]. Available: https://w3.org/TR/rdf-concepts/

[135] Z. Xu, M. Karlsson, C. Tang, and C. T. Karamanolis, "Towards a Semantic-Aware File Store," *Usenix HotOS*, pp. 181–187, 2003.

[136] T. White, *Hadoop: The Definitive Guide.* O'Reilly Media, Inc., 2012.

[137] Red Hat, Inc. Gluster. Last accessed on 21/04/2018. [Online]. Available: https://gluster.org/

[138] The Apache Software Foundation. HDFS Architecture. Last accessed on 19/04/2018. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[139] K. McKusick and S. Quinlan, “GFS: Evolution on Fast-Forward,” *Communications of the ACM*, vol. 53, no. 3, pp. 42–49, 2010.

[140] A. Fikes, “Storage Architecture and Challenges,” *Talk at the Google Faculty Summit*, 2010.

[141] The Apache Software Foundation. Apache Hadoop Releases. Last accessed on 21/04/2018. [Online]. Available: http://hadoop.apache.org/releases.html

[142] Apple, Inc. How to Use Time Machine to Back Up or Restore your Mac. Last accessed on 23/04/2018. [Online]. Available: https://support.apple.com/en-gb/HT201250

[143] Free Software Foundation, Inc. Concurrent Versions System. Last accessed on 23/04/2018. [Online]. Available: http://savannah.nongnu.org/projects/cvs

[144] Canonical, Ltd. Bazaar. Last accessed on 23/04/2018. [Online]. Available: http://bazaar.canonical.com

[145] R. C. Merkle, “Secrecy, Authentication, and Public Key Systems,” Ph.D. dissertation, Stanford University, 1979.

[146] R. C. Merkle, “Method of Providing Digital Signatures,” Jan. 5 1982, US Patent 4,309,569.

[147] S. Chacon and B. Straub, *Pro git*. Apress, 2014, Last accessed on 05/02/2018. [Online]. Available: https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History

[148] G. Gore. (2017, Dec.) Glenn’s take on re:invent 2017 – part 3. Last accessed on 05/04/2018. [Online]. Available: https://aws.amazon.com/blogs/architecture/glenns-take-on-reinvent-2017-part-3/

[149] Amazon.com, Inc. Object Versioning. Last accessed on 27/03/2018. [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/dev/ObjectVersioning.html

[150] Amazon.com, Inc. AWS Identity and Access Management (IAM). Last accessed on 08/04/2018. [Online]. Available: https://aws.amazon.com/iam/

[151] Amazon.com, Inc. Managing Access Permissions to Your Amazon S3 Resources. Last accessed on 08/04/2018. [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/dev/s3-access-control.html

[152] Amazon.com, Inc. Protecting Data in Amazon S3. Last accessed on 05/04/2018. [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/dev/DataDurability.html

[153] Amazon.com, Inc. Amazon S3 Storage Classes. Last accessed on 04/04/2018. [Online]. Available: https://aws.amazon.com/s3/storage-classes/

[154] Amazon.com, Inc. Amazon Drive. Last accessed on 28/11/2017. [Online]. Available: https://amazon.co.uk/clouddrive

[155] Apple, Inc. iCloud. Last accessed on 28/11/2017. [Online]. Available: https://icloud.com

[156] Adobe Systems, Inc. Adobe Creative Cloud. Last accessed on 27/03/2018. [Online]. Available: https://adobe.com/uk/creativecloud.html

[157] R. Armstrong. (2017, Sep.) DBX Platform Launches with Three New APIs. Last accessed on 01/01/2018. [Online]. Available: https://blogs.dropbox.com/developers/2017/09/dbx-platform-launches-with-three-new-apis/

[158] W. van der Laan. (2013) Dropship. Last accessed on 04/04/2018. [Online]. Available: https://github.com/driverdan/dropship

[159] Dropbox, Inc. File Version History. Last accessed on 24/04/2018. [Online]. Available: https://dropbox.com/help/security/version-history-overview

[160] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside Dropbox: understanding Personal Cloud Storage Services," *Proceedings of the 2012 Internet Measurement Conference*, pp. 481–494, 2012.

[161] K. Modzelewski. (2012) How we've scaled Dropbox. Youtube - Stanford University. Last accessed on 10/03/2018. [Online]. Available: https://youtube.com/watch?v=PE4gwstWhmc

[162] A. Gupta. (2016, Mar.) Scaling to Exabytes and Beyond. Last accessed on 10/01/2018. [Online]. Available: https://blogs.dropbox.com/tech/2016/03/magic-pocket-infrastructure/

[163] D. Gupta. (2016, Aug.) (Re)Introducing Edgestore. Last accessed on 20/04/2018. [Online]. Available: https://blogs.dropbox.com/tech/2016/08/reintroducing-edgestore/

[164] J. Cowling. (2016, May) Inside the Magic Pocket. Last accessed on 04/04/2018. [Online]. Available: https://blogs.dropbox.com/tech/2016/05/inside-the-magic-pocket/

[165] M. Dee. (2015, Oct.) Inside LAN Sync. Last accessed on 13/03/2018. [Online]. Available: https://blogs.dropbox.com/tech/2015/10/inside-lan-sync/

[166] Google, LLC. Introducing New, Enterprise-Ready Tools for Google Drive. Last accessed on 14/04/2018. [Online]. Available: https://blog.google/products/g-suite/introducing-new-enterprise-ready-tools-google-drive/

[167] S. E. Lindley, G. Smyth, R. Corish, A. Loukianov, M. Golembewski, E. A. Luger, and A. Sellen, "Exploring New Metaphors for a Networked World through the File Biography," *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, p. 118, 2018.

[168] O. Goldman, R. Horns, S. Switzer, J. Wixson, and H. Khalfallah, "Digital Composites: Technology for Creativity in a Cloud-Connected World," *Internet Technologies and Applications*, pp. 144–150, 2015.

[169] ODrive. ODrive - Sync all Cloud Storage in One Place. Last accessed on 02/04/2018. [Online]. Available: https://odrive.com/

[170] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DepSky: Dependable and Secure Storage in a Cloud-of-Clouds," *ACM Transactions on Storage*, vol. 9, no. 4, p. 12, 2013.

[171] K. D. Bowers, A. Juels, and A. Oprea, "HAIL: A High-Availability and Integrity Layer for Cloud Storage," *Proceedings of the 16th ACM Conference on Computer and Communications Security*, pp. 187–198, 2009.

[172] E. Stefanov and E. Shi, "Multi-Cloud Oblivious Storage," *Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security*, pp. 247–258, 2013.

[173] T. G. Papaioannou, N. Bonvin, and K. Aberer, "Scalia: an Adaptive Scheme for Efficient Multi-Cloud Storage," *International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–10, 2012.

[174] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, "RACS: a Case for Cloud Storage Diversity," *Proceedings of the 1st ACM Symposium on Cloud Computing*, pp. 229–240, 2010.

[175] Y. Singh, F. Kandah, and W. Zhang, "A Secured Cost-Effective Multi-Cloud Storage in Cloud Computing," *IEEE Conference on Computer Communications Workshops*, pp. 619–624, 2011.

[176] R. K. Bedi, J. Singh, and S. K. Gupta, "A Novel Approach for Multi-Cloud Storage for Mobile Devices," *International Journal of Information Technology and Web Engineering*, vol. 13, no. 2, pp. 24–36, 2018.

[177] J. Liu, E. Ahmed, M. Shiraz, A. Gani, R. Buyya, and A. Qureshi, "Application Partitioning Algorithms in Mobile Cloud Computing: Taxonomy, Review and Future Directions," *Journal of Network and Computer Applications*, vol. 48, pp. 99–117, 2015.

[178] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A Survey and Comparison of Peer-to-Peer Overlay Network Schemes," *IEEE Communications Surveys and Tutorials*, vol. 7, no. 2, pp. 72–93, 2005.

[179] M. Ripeanu, "Peer-to-Peer Architecture Case Study: Gnutella Network," *First International Conference on Peer-to-Peer Computing*, pp. 99–100, 2001.

[180] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A Distributed Anonymous Information Storage and Retrieval System," *Designing Privacy Enhancing Technologies*, pp. 46–66, 2001.

[181] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a Scalable Peer-to-Peer Lookup Protocol for Internet Applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, 2003.

[182] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A Scalable Content-Addressable Network.* ACM, 2001, vol. 31, no. 4.

[183] A. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pp. 329–350, 2001.

[184] B. Y. Zhao, J. Kubiatowicz, and A. D. Joseph, "Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing," *Technical Report, Computer Science Division, University of California Berkeley*, 2001, Last accessed on 27/04/2018.

[Online]. Available: https://people.eecs.berkeley.edu/~adj/publications/paper-files/CSD-01-1141.pdf

[185] P. Maymounkov and D. Mazières, “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric,” *International Workshop on Peer-to-Peer Systems*, pp. 53–65, 2002.

[186] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, “Looking up Data in P2P Systems,” *Communications of the ACM*, vol. 46, no. 2, pp. 43–48, 2003.

[187] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, “Ivy: A Read/Write Peer-to-Peer File System,” *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 31–44, 2002.

[188] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, “Wide-Area Cooperative Storage with CFS,” *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 202–215, 2001.

[189] J.-M. Busca, F. Picconi, and P. Sens, “Pastis: A Highly-Scalable Multi-User Peer-to-Peer File System,” *European Conference on Parallel Processing*, pp. 1173–1182, 2005.

[190] Perkeep Community. Perkeep (previously Camlistore). Last accessed on 25/04/2018. [Online]. Available: https://perkeep.org/

[191] The Tahoe-LAFS Developers. (2016) Mutable Files - Tahoe-LAFS. Last accessed on 18/04/2018. [Online]. Available: http://tahoe-lafs.readthedocs.io/en/tahoe-lafs-1.12.1/specifications/mutable.html

[192] M. Ogden, “Dat – Distributed Dataset Synchronization And Versioning,” *Open Science Framework*, 2017, Last accessed on 23/05/2018. [Online]. Available: https://datproject.org/paper

[193] B. Warner, Z. Wilcox-O'Hearn, and R. Kinninmont. (2008) Tahoe: A Secure Distributed Filesystem. Last accessed on 11/03/2018. [Online]. Available: http://allmydata.org/~warner/pycon-tahoe.html

[194] S. C. Rhea, P. R. Eaton, D. Geels, H. Weatherspoon, B. Y. Zhao, and J. Kubiatowicz, "Pond: The OceanStore Prototype," *Usenix FAST*, vol. 3, pp. 1–14, 2003.

[195] C. Wells, "The OceanStore Archive: Goals, Structures, and Self-Repair," *Master's thesis, UC Berkeley*, 2000.

[196] D. M. Geels, "Data Replication in OceanStore," *Technical Report, University of California at Berkeley*, 2002, Last accessed on 17/03/2018. [Online]. Available: http://oceanstore.cs.berkeley.edu/publications/papers/pdf/geels-masters.pdf

[197] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel, "Separating Key Management from File System Security," *ACM SIGOPS Operating Systems Review*, vol. 33, no. 5, pp. 124–139, 1999.

[198] Protocol Labs. IPLD – InterPlanetary Linked Data. Last accessed on 04/03/2018. [Online]. Available: https://ipld.io/

[199] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008, Last accessed on 23/04/2018. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[200] Ethereum Community. Ethereum White Paper: A Next-Generation Smart Contract and Decentralized Application Platform. Last accessed on 28/04/2018. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[201] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, "Zcash Protocol Specification," *Technical report, Version 2018.0–Beta–22. Zerocoin Electric Coin Company*, 2016, Last accessed on 20/07/2018. [Online]. Available: https://github.com/zcash/zips/raw/master/protocol/protocol.pdf

[202] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips, "The Bittorrent P2P File-Sharing System: Measurements and Analysis," *International Workshop on Peer-to-Peer Systems*, pp. 205–216, 2005.

[203] D. Mazières, "Self-certifying File System," Ph.D. dissertation, Massachusetts Institute of Technology, 2000.

[204] A. K. Dey, "Understanding and Using Context," *Personal and Ubiquitous Computing*, vol. 5, no. 1, pp. 4–7, 2001.

[205] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles, "Towards a Better Understanding of Context and Context-Awareness," *International Symposium on Handheld and Ubiquitous Computing*, pp. 304–307, 1999.

[206] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. J. O'Toole, "Semantic File Systems," *ACM SIGOPS Operating Systems Review*, vol. 25, no. 5, pp. 16–25, 1991.

[207] E. Freeman and D. Gelernter, "Lifestreams: A Storage Model for Personal Data," *ACM SIGMOD Record*, vol. 25, no. 1, pp. 80–86, 1996.

[208] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt, "A Middleware Infrastructure for Active Spaces," *IEEE Pervasive Computing*, vol. 1, no. 4, pp. 74–83, 2002.

[209] K. Veeraraghavan, J. Flinn, E. B. Nightingale, and B. Noble, "quFiles: The Right File at the Right Time," *ACM Transactions on Storage*, vol. 6, no. 3, p. 12, 2010.

[210] S. Khungar and J. Riekki, "Context Based Storage: System for Managing Data in Ubiquitous Computing Environment," *Proceedings of the 12th International Conference on Advanced Computing and Communication*, 2004.

[211] Y. Hua, H. Jiang, Y. Zhu, D. Feng, and L. Tian, "SmartStore: A New Metadata Organization Paradigm with Semantic-Awareness for Next-Generation File Systems,"

Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, p. 10, 2009.

[212] Y. Hua, H. Jiang, Y. Zhu, D. Feng, and L. Xu, "SANE: Semantic-Aware Namespace in Ultra-Large-Scale File Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1328–1338, 2014.

[213] A. K. Dey and G. D. Abowd, "Cybreminder: A Context-Aware System for Supporting Reminders," *International Symposium on Handheld and Ubiquitous Computing*, pp. 172–186, 2000.

[214] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger, "Perspective: Semantic Data Management for the Home," *Usenix FAST*, vol. 9, pp. 167–182, 2009.

[215] N. Bila, T. Ronda, I. Mohomed, K. N. Truong, and E. de Lara, "Pagetailor: Reusable End-User Customization for the Mobile Web," *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, pp. 16–29, 2007.

[216] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir, "Adapting to Network and Client Variability via On-Demand Dynamic Distillation," *ACM SIGOPS Operating Systems Review*, vol. 30, no. 5, pp. 160–170, 1996.

[217] J. Flinn and M. Satyanarayanan, "Energy-Aware Adaptation for Mobile Applications," *Proceedings of the 17th ACM symposium on Operating Systems Principles*, vol. 33, no. 5, 1999.

[218] I. Sodagar, "The MPEG-DASH Standard for Multimedia Streaming over the Internet," *IEEE MultiMedia*, vol. 18, no. 4, pp. 62–67, 2011.

[219] J. R. Douceur, "The Sybil Attack," *International Workshop on Peer-to-Peer Systems*, pp. 251–260, 2002.

[220] P. Rogaway and T. Shrimpton, "Cryptographic Hash-function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance,

and Collision Resistance," *International wWrkshop on Fast Software Encryption*, pp. 371–388, 2004.

[221] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen, "HTTP Extensions for Distributed Authoring – WEBDAV," Internet Requests for Comments, RFC Editor, RFC 2518, February 1999, Last accessed on 20/07/2018. [Online]. Available: https://rfc-editor.org/rfc/rfc2518.txt

[222] M. Tauber, "Autonomic Management in a Distributed Storage System," Ph.D. dissertation, University of St Andrews, 2010.

[223] Protocol Labs. Multiformats. Last accessed on 04/03/2018. [Online]. Available: https://multiformats.io/

[224] Oracle, Corp. Getting Started with the G1 Garbage Collector. Last accessed on 04/05/2018. [Online]. Available: http://oracle.com/technetwork/tutorials/tutorials-1876574.html

[225] D. Vorick and L. Champine, "Sia: Simple Decentralized Storage," *Technical report, NebulousLabs, Boston (MA), USA*, 2014, Last accessed on 05/04/2018. [Online]. Available: https://sia.tech/sia.pdf

[226] S. Wilkinson, T. Boshevski, J. Brandoff, J. Prestwich, G. Hall, P. Gerbes, P. Hutchins, and C. Pollard, "Storj: A Peer-to-Peer Cloud Storage Network," *Technical report, Storj Labs Inc., Atlanta (GA), USA*, Last accessed on 16/04/2018. [Online]. Available: https://storj.io/storj.pdf

[227] Protocol Labs, "Filecoin: A Decentralized Storage Network," *Technical report, Protocol Labs*, 2017, Last accessed on 30/05/2018. [Online]. Available: https://filecoin.io/filecoin.pdf

[228] N. Szabo, "Smart Contracts: Building Blocks for Digital Markets," *EXTROPY: The Journal of Transhumanist Thought*, 1996.

[229] A. Dearle, G. Kirby, S. Norcross, and A. McCarthy, “A Peer-to-Peer Middleware Framework for Resilient Persistent Programming,” *arXiv preprint arXiv:1006.3724*, 2010.

Image Credits

The images and diagrams used in this thesis are my own creation. Some of the diagrams in the *Background* and *Literature Review* Chapters are inspired from others' work, which have all been cited.

The following images have been used for the creation of some of the diagrams:

- The *Orange Fish* image was retrieved from: `http://firstaidforfree.com/what-does-fish-shaped-stand-for-in-first-aid/` [last accessed on 14/02/2018].

- The *Blue Fish* image was retrieved from: `https://mysticmorning.deviantart.com/art/Blue-Fish-png-292103542` [last accessed on 14/02/2018].

- The *Cat* image was retrieved from: `https://petmd.com/cat/behavior/evr_ct_what-does-it-mean-when-a-cat-wags-tail` [last accessed on 26/04/2018].

- The *Laptop* icon was retrieved from: `https://thenounproject.com/term/laptop/66295/` [last accessed on 14/02/2018].

- The *Desktop PC* icon was retrieved from: `http://myiconfinder.com/icon/computer-desktop-device-digital-display-electronic-monitor-pc-cpu/9150` [last accessed on 14/02/2018].

- The *Document* icon was retrieved from: `https://flaticon.com/free-icon/text-document_32329` [last accessed on 14/02/2018].

- The *Apple Time Machine* screenshot was retrieved from: `https://support.apple.com/en-gb/HT201250` [last accessed on 16/03/2018].

- The *Key* icon was retrieved from: `https://flaticon.com/free-icon/black-key-horizontal-shape_44549` [last accessed on 17/04/2018].

List of Abbreviations

Abbreviations are ordered alphabetically.

General Terms

AES	Advanced Encryption Standard
CAP	Consistency, Availability, Partition Tolerance
CFS	Clustered File Systems
CoC	Cloud of Clouds
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
CSFS	Client-Server File System
DAG	Directed Acyclic Graph
DES	Data Encryption Standard
DFS	Distributed File System
DMS	Data Management System
DOI	Digital Object Identifier
DSA	Digital Signature Algorithm
EEM	Explicit Extrinsic Metadata
EIM	Explicit Intrinsic Metadata
FCB	File-Control Block
GUID	Globally Unique Identifier
IEM	Implicit Extrinsic Metadata
IIM	Implicit Intrinsic Metadata
MD5	Message Digest 5
HTTP	Hypertext Transfer Protocol

IP	Internet Protocol
P2P	Peer-to-Peer
JSON	JavaScript Object Notation
IO or I/O	Input/Output
RAM	Random-access memory
OS	Operating System
NAS	Network-Attached Storage
NTP	Network Time Protocol
SAN	Storage-Area Network
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
SaaS	Software as a Service
MITM	Man-in-the-Middle
SNFS	Shared-Nothing File System
VFS	Virtual File System
VCS	Version Control System
SHA	Secure Hash Algorithm
SPOF	Single Point of Failure
RC4	Rivest Cipher 4
RSA	Rivest-Shamir-Adleman
WebDAV	Web Distributed Authoring and Versioning
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

Systems and Tools

AFS	Andrew File System
Amazon S3	Amazon Simple Storage Service
CIFS	Common Internet File System
DBFS	Oracle Database File System
FUSE	Filesystem in Userspace
GFS	Google File System
IFS	Installable File System
IPFS	InterPlanetary File System
NFS	Network File System
OCFS2	Oracle Cluster File System
Windows NTFS	Windows New Technology File System
Windows FAT	Windows File Allocation Table
SFS	Semantic File System
SMB	Server Message Block

Abbreviations Regarding the Model Introduced in this Thesis

CMS	Context Management Service
NMS	Node Management Service
MDMS	Manifests and Data Management Service
MMS	Metadata Management Service
SMS	Storage Management Service
SOS	Sea Of Stuff
URMS	User and Role Management Service

Abbreviations Used in the Experimental Chapters

BWC	Butts Wynd Cluster
EFU	Experimental Framework Utility

Abbreviations regarding the datasets used for the experiments performed to evaluate this work are introduced in the appropriate sections of Chapter 8.

Glossary of Terms

The following is a short glossary of terms that will be used throughout this thesis.

- **Abstraction**: the generic essence of a concept. In this thesis the word abstraction is used to indicate the generic concepts to model data, such as the file or the directory. The word abstraction is used interchangeably with the term *metaphor*.
- **Cardinality** of a set: the number of entities that make the set.
- **Content-addressable**: a system where data is associated with names that are related to its actual sequence of bytes, such that a different sequence of bytes must be bound to another name.
- **Immutable**: unchanging over time. An object or entity that is immutable cannot change its state once it is created.
- **Node**: a point of access to a network. A node must be uniquely identifiable within the network (*e.g.*, via IP address).
- **Metaphor**: used interchangeably with the term *abstraction*.
- **Provenance**: the history of an entity. The provenance of an entity consists of metadata about its creation and how it has changed over time, by whom, where, why, under what condition, and so on.
- **Re-computable**: having the capability to be computed again given some data.
- **Self-describing**: containing all the information necessary to characterise itself such that it does not need any further external data to be used in the system.

- **Self-contained**: see *self-describing*.

- **Size** of a set: the sum of the size (*e.g.*, in bytes) of all entities in the set.

- **Transparency**: transparency is an ambiguous word, since it can have two meanings: (1) there is transparency in some aspect of a distributed system when this is hidden from the user or (2) there is transparency is when an aspect of the system is visible. For the purpose of this thesis, the definition in (1) will be used.

Appendices

Settings

A.1 SOS Node Settings

The following is an example of a setting file for a SOS node. The format is in JSON.

```
{
  "settings": {
    "guid": "SHA256_16_9999a025d7d3b2cf782da0ef24423181fdd4096091bd8cc18b18c3aab9cb00bb",
    "services": {
      "sms": {
        "exposed": true,
        "canPersist": true,
        "maxReplication": 3
      },
      "cms": {
        "exposed": false,
        "indexFile": "cms.index",
        "loadedPath": "~/sos/java/contexts/",
        "automatic": true,
        "predicateThread": {
          "initialDelay": 30,
          "period": 60

        },
        "policiesThread": {
          "initialDelay": 45,
          "period": 60
        },
        "checkPoliciesThread": {
          "initialDelay": 45,
          "period": 60
      },
      "getdataThread":  {
        "initialDelay":  60,
        "period":  60
      },
      "spawnThread":  {
        "initialDelay":  90,
        "period":  120
      }
    },
    "mdms":  {
      "exposed":  false,
      "maxReplication":  3,
      "cacheFile":  "manifests.cache",
      "indexFile":  "dds.index"
    },
    "urms":  {
      "exposed":  false,
      "cacheFile":  "usro.cache"
    },
    "nms":  {
      "exposed":  false,
      "startupRegistration" : true,
      "bootstrap" : true,
      "ping":  true
    },
    "mms":  {
      "exposed":  false
    }
  },
  "database":  {
    "filename":  "node.db"
  },
  "rest":  {
    "port":  8080
  },
  "webDAV":  {
    "port":  8081
  },
  "webAPP":  {
      "port":  8082
    },
    "keys":  {
      "location":  "~/sos/keys"
    },
    "store":  {
      "type":  "local",
      "location":  "~/sos/"
    },
    "global":  {
      "ssl_trust_store" : "PATH TO THE JAVA SECURITY CACERTS",
      "tasks":  {
        "thread":  {
          "ps":  4
        }
      },
      "cacheFlusher":  {
        "enabled" : false,
        "maxSize":  1048576,
        "thread":  {
          "ps":  1,
          "initialDelay":  0,
          "period":  600
        }
      }
    },
    "bootstrapNodes":  [
      {
      "guid" : "SHA256_16_bb077f9420219e99bf776a7a116334405a81d2627bd4f87288259607f05d1615",
      "hostname" : "138.251.207.87",
      "port" : 8080,
      "certificate" : "MFwwDQYJKoZIhvcNAQEBBQADSwAwSAJBAKZ0noFAxsx4BiXBKzeJIS0v5q5XTSpPZRCmYGg+
           59VctY1xeYS7NEkEmbk/Sa8y5chrZttN5CggdBJBIFGgMU0CAwEAAQ=="
      }
    ]
  }
}
```

Code A.1: Example of SOS node setting file.

Experiment Settings

B.1 Butts Wynd Cluster

The following is the specification for the OS used by the nodes of the Butts Wynd Cluster (relative to the time when the experiments were run).

```
# Kernel Version
$ uname -r
Linux 3.10.0-693.11.6.el7.x86_64

# Linux Distribution
$ cat /etc/*-release
NAME="Scientific Linux"
VERSION="7.4 (Nitrogen)"
ID="rhel"
ID_LIKE="scientific centos fedora"
VERSION_ID="7.4"
PRETTY_NAME="Scientific Linux 7.4 (Nitrogen)"
ANSI_COLOR="0;31"
CPE_NAME="cpe:/o:scientificlinux:scientificlinux:7.4:GA"
HOME_URL="http://www.scientificlinux.org//"
BUG_REPORT_URL="mailto:scientific-linux-devel@listserv.fnal.gov"

REDHAT_BUGZILLA_PRODUCT="Scientific Linux 7"
REDHAT_BUGZILLA_PRODUCT_VERSION=7.4
REDHAT_SUPPORT_PRODUCT="Scientific Linux"
REDHAT_SUPPORT_PRODUCT_VERSION="7.4"
Scientific Linux release 7.4 (Nitrogen)
Scientific Linux release 7.4 (Nitrogen)
Scientific Linux release 7.4 (Nitrogen)
```

Code B.1: Kernel Version and Operating System Version.

B.2 Experiment Configuration

An experiment is defined by a Java class, which matches the experiment phases described in Section 8.4, and an experiment configuration file. The experiment configuration file contains information about the experiment itself and the nodes where to distribute it. The following is an example of such file.

```
{
  "experiment" : {
    "name" : "experiment pr",
    "experimentClass": "Experiment_PR", // Must match the actual Java file for the expriment
    "description" : "PR experiment - on hogun-2.",
    "setup" : {
      "app" : "sos-slave/target/sos.jar",
      "iterations" : 10
    },
    "slave_nodes" : [ ],
    "coordinator_node" : {
      "id" : 0,
      "name" : "hogun-2",
      "path" : "/cs/scratch/", // Remote path
      "remote" : true,
      "java" : "/usr/local/jdk/bin/java", // Remote path
      "ssh" : {
        "type" : 1,
        "host" : "hogun-2.cluster",
        "user" : "sic2",
        "known_hosts": "/Users/sic2/.ssh/known_hosts",
        "config": "/Users/sic2/.ssh/config",
        "privatekeypath": "/Users/sic2/.ssh/id_rsa",
        "passphrase" : "ENCRYPTED_PASSPHRASE"
      },
      "configurationfile" : "NODE_CONFIGURATION_FILE",
      "dataset" : "NAME_OF_DATASET",
    },
    "stats": { // Toggles to enable/disable instrumentation of the SOS node
      "experiment": true,
      "predicate": true,
      "predicate_remote": false,
      "policies": false,
      "checkPolicies": false,
      "checkPolicies": false,
      "io": false,
      "guid_data": false,
      "guid_manifest": false,
      "ping": false
    }
  }
}
```

Code B.2: Kernel Version and Operating System Version.

Libraries used by the SOS prototype

This appendix section presents the most relevant libraries used by the SOS prototype. Some of these libraries have been developed as part of this thesis, while others have been implemented by third parties.

The SOS prototype does not use any proprietary library/code.

C.1 Castore

Castore (cast + store) is a very minimal and simple library that provides abstraction over different storage systems. Castore, currently, provides a storage abstraction over the following implementations: local file system, AWS S3, Redis, Dropbox, and Google Drive.

The castore library is inspired by the **asa/filesystem** project developed at the University of St Andrews [229].[114]

A SOS node can be configured to use different underlying storage architectures, since all the in-node data interactions are done through the castore library.

The library is available at `https://github.com/sea-of-stuff/castore` (MIT).

C.2 guid-sta

This is a simple library to generate GUIDs using the format described in Section 5.1.1. This library is available at `https://github.com/stacs-srg/guid-sta` (MIT).

[114] Github - stacs-srg/asa. `https://github.com/stacs-srg/asa` [Last accessed on 15/05/2018].

C.3 St Andrews Utilities

This is an effort of the University of St Andrews system research group to collect common utility algorithms written in *java* written over the years and across multiple projects. This library is available at `https://github.com/stacs-srg/utilities` (GPL-3.0).

C.4 Apache Tika

The Apache Tika™dependency is used to automatically analyse atoms and generate metadata for the SOS. This library was chosen because of its ease of usage and very wide data type support. The Apache Tika™library is available at `https://tika.apache.org/`.

C.5 Unirest

Unirest is a lightweight HTTP library that allows the writing of HTTP requests in a simple manner. The SOS prototype relies on Unirest for the communication between nodes. Unirest is available at `http://unirest.io/` (MIT).

C.6 WebDAV-server

This is a java-based WebDAV server prototype that has been used by several research projects to expose different types of storage implementations. The WebDAV-server project is available at: `https://github.com/stacs-srg/WebDAV-server`.

C.7 fs-sta

The fs-sta project consists of a collection of generic file system abstractions that have been shared across multiple research projects. The SOS implements these abstractions to support the WebDAV file system. The fs-sta abstractions are available at: `https://github.com/stacs-srg/fs-sta`.

Systems Comparison

The tables in the next two pages provide a summary of the comparisons presented in Chapter 7 in a tabular form.

Information that was not available or not applicable has been marked as “N/A”.

		Local File Systems	**Networked File Systems**					**Versioning Storage Systems**		
	SOS	**(ext2-3-4, HFS, FAT, etc)**	**AFS**	**NFS**	**SMB/CIFS**	**Google File System**	**Hadoop**	**Apple Time Machine**	**git**	**mercurial**
Data representation	Atom	File	File	File	File	File	File	File	Blob	File
Data Aggregation	Compound	Directory	Directory	Directory	Directory	None	None	Directory	Tree	Manifest
Location Abstraction	Unstructured P2P (but can support DHT)	No	Yes	Yes	Yes	Yes	Yes	N/A	nodes specified in configuration	nodes specified in configuration
Data Type	Immutable	Mutable	Mutable	Mutable	Mutable	Append-only	Append-only	Mutable	Immutable + staging phase (mutable)	Immutable + staging phase (mutable)
De-duplication	Yes	No	No	No	No	Yes	Yes	Yes, but not across all files	Yes	No (based on deltas)
Entities Internal Identifiers	GUID (SHA256 by default)	Human readable name	Human readable name	Human readable name	Human readable name	Human readable name + Hash	Human readable name + Hash	Human readable name	SHA-1 (migrating to SHA-256)	Numbers (local to repo) and SHA-1 (for revlog, < 0.9v) and SHA-256 (for revlogNG, >= 0.9)
Data Integrity	Yes	Depends on the implementation	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Erasure Coding	Via Contexts	No	No	No	No	Yes	Yes	No	No	No
Data replication	Via Contexts	N/A	Yes	Yes	Yes	Yes	Yes	No	Manually	Manually
Metadata Support	Yes, external metadata model	Yes, via attributes, extended attributes, and tagging	Yes, via attributes	Yes, via attributes	Yes, via attributes	N/A	N/A	Yes, via attributes	Yes, via commits and notes	Yes, via commits
Versioning Support	Yes	No	No	No	No	No	No	Yes	Yes	Yes
Network Protocol	HTTP	No	TCP	UDP and TCP	SMB	N/A	N/A	AFP, SMB	git (w/wo ssh), HTTP, HTTPS, rsync, email, bundles	Custom protocol, HTTP, email
User model	User-Role model	User, group, other model, ACL, Capabilities	User, group, other model, ACL, Capabilities	User, group, other model, ACL, Capabilities	User, group, other model, ACL	Yes	Yes	No	Yes (committer/author)	Yes (commiter)
Data Protection	Yes	Depends	In-transit	In-transit	In-transit	Yes	Yes	No	No	No
Automatic File Classification	Via Contexts	No	No	No	No	No	No	No	No	No
Smart folders	Via Contexts	No	No	No	No	No	No	No	No	No
Automatic File Management	Via Contexts	No	No	No	No	Yes	Yes	No	No	No

		P2P				SaaS Storage	IaaS Storage	Context-aware Storage			
	SOS	OceanStore/Pond	IPFS	Perkeep	Tahoe-LAFS	Dropbox	Amazon S3	Semantic FS	quFiles	Microsoft File Biography	Creative Cloud
Data representation	Atom	Object (active and archival)	Block	File and bytes (collection of blobs)	File	File	Object	File	quFile	File Biography	File, DCX
Data Aggregation	Compound	Supported, object known simply as directory	List and Tree	Directory	Directory	Directory	None	Virtual Directory	Directory	Directory	Directory
Location Abstraction	Unstructured P2P (but can support DHT)	DHT - Tapestry (initial design was based on unstructured P2P)	DHT - Kademlia	Yes	Ad-hoc server selection	Yes	Yes	Yes	Yes	Yes	Yes
Data Type	Immutable	Read-only data (immutable) and changeable data (mutable)	Immutable	Immutable	Immutable and Mutable	Mutable	immutable	Mutable	Mutable	Immutable	Mutable
De-duplication	Yes	Yes	Yes	Yes	Yes (with convergence secret)	Yes (per namespace)	Yes	No	No	N/A	N/A
Entities Internal Identifiers	GUID (SHA256 by default)	SHA-1 or hash(name + pub_key) for mutable data	Multiformat	SHA-256 (with blobref format)	SHA-256 (called capabilities)	Human readable name + Hash	Object name + Version id	Human readable name	Human readable name	N/A	Human readable name + Hash
Data Integrity	Yes	Yes	Yes	Yes	Yes	Yes	Yes	N/A	N/A	N/A	N/A
Erasure Coding	Via Contexts	Active data and archival data	No	No	Yes (only 3 blocks out of 10 are needed to reconstruct a segment with the default settings)	Yes at bucket level	N/A	N/A	N/A	N/A	N/A
Data replication	Via Contexts	Yes	Via protocols built on top of it	Yes	Yes	On server-side	On server-side	No	Via policies	No	On server-side
Metadata Support	Yes, external metadata model	No	No	Yes, via attribute claims	Yes, represented as the edges of the graph	Yes, stored via Edgestore	Yes, attributes and tags	Yes, via attributes	Yes, via attributes	Yes	Yes, via attributes and embedded in DCX format
Versioning Support	Yes	Yes	Yes	Yes	N/A	Yes	Yes	No	No	Yes	Yes
Network Protocol	HTTP	Tapestry on top of TCP	Bitswap, HTTP	HTTP	TCP	UDP, HTTPS	HTTPS	N/A	N/A	N/A	HTTPS
User model	User-Role model	ACL	N/A	N/A	Yes (capabilities)	Yes	Yes	N/A	N/A	N/A	Yes
Data Protection	Yes	Yes	In-transit	No	At-rest	On server-side	On server-side	N/A	N/A	N/A	On server-side
Automatic File Classification	Via Contexts	No	No	No	No	No	No	Yes	Yes, via policies	N/A	No
Smart folders	Via Contexts	No	No	Yes	No	No	No	Yes	Yes, via policies	N/A	Yes
Automatic File Management	Via Contexts	No	No	Data importers only	No	Yes	Y (can set policies based on tags and prefix)	No	Yes, via policies	Yes	Yes