TClouds

SEVENTH FRAMEWORK PROGRAMME

# D2.2.2
# Preliminary Specification of Services and Protocols of Middleware for Adaptive Resilience

| | |
|---|---|
| **Project number:** | 257243 |
| **Project acronym:** | TClouds |
| **Project title:** | Trustworthy Clouds - Privacy and Resilience for Internet-scale Critical Infrastructure |
| **Start date of the project:** | 1st October, 2010 |
| **Duration:** | 36 months |
| **Programme:** | FP7 IP |

| | |
|---|---|
| **Deliverable type:** | Report |
| **Deliverable reference number:** | ICT-257243 / D2.2.2 / 1.0 |
| **Activity and Work package contributing to deliverable:** | Activity 2 / WP 2.2 |
| **Due date:** | September 2012 – M24 |
| **Actual submission date:** | 28th September, 2012 |

| | |
|---|---|
| **Responsible organisation:** | FFCUL |
| **Editor:** | Alysson Bessani |
| **Dissemination level:** | Public |
| **Revision:** | 1.0 (r7553) |



| | |
|---|---|
| **Abstract:** | This document describes a revised TClouds architecture and how dependable services can use adaptation on the cloud. Besides that, it describes a number of components and algorithms developed during the first two years of the project. |




| | |
|---|---|
| **Keywords:** | Architecture, resilience, adaptation, cloud-of-clouds, object storage, replication, Byzantine fault tolerance, coordination |

**Editor**

Alysson Bessani (FFCUL)

**Contributors**

Fernando André, Alysson Bessani, Vinícius Cogo, Marcelo Pasin, Bruno Quaresma, João Sousa, Paulo Sousa and Paulo Veríssimo (FFCUL)

Cristina Băsescu, Christian Cachin, Ittay Eyal, Robert Haas, Birgit Junker, Nikola Knežević, Alessandro Sorniotti and Marko Vukolić (IBM)

Rüdiger Kapitza (TUBS)

Tobias Distler (FAU)

**Disclaimer**

This work was partially supported by the European Commission through the FP7-ICT program under project TClouds, number 257243.

The information in this document is provided as is, and no warranty is given or implied that the information is fit for any particular purpose.

The user thereof uses the information at its sole risk and liability. The opinions expressed in this deliverable are those of the authors. They do not necessarily represent the views of all TClouds partners.

# Executive Summary

In this deliverable we describe our 2nd-year efforts in further developing the cloud-of-clouds model, in which a set of cloud providers are used to build services such as object storage (in the same line as object-based cloud storage IaaS) and replicated execution (something similar to a PaaS). We start by providing a consolidated description of the rationale and aims of the TClouds reference architecture, and then we proceed to a discussion about the need for adaptation in cloud-based services. The remaining of the document describes several contributions of the project regarding how to replicate, coordinate and access computation and information located in several independent and non-cooperating clouds. All these contributions are related to a subset of components/ideas provided in the first year. In particular, we focus on three main components: cloud-of-clouds object storage (theory and practice), Byzantine fault-tolerant state machine replication and fault-tolerant BPEL and its efficient implementation.

# Contents

**1 Introduction** **1**
1.1 TClouds — Trustworthy Clouds . . . 1
1.2 Activity 2 — Trustworthy Internet-scale Computing Platform . . . 1
1.3 Workpackage 2.2 — Cloud of Clouds Middleware for Adaptive Resilience . . . 2
1.4 Deliverable 2.2.2 — Preliminary Specification of Services and Protocols of Middleware for Adaptive Resilience . . . 2

**2 Revised TClouds Architecture** **6**
2.1 Introduction . . . 6
2.2 Motivation and Approach . . . 7
2.3 Architecture Overview . . . 9
2.4 Main Building Blocks . . . 10
2.5 Examples of TClouds distributed middleware . . . 11
2.6 Conclusion . . . 14

**3 Adaptation** **15**
3.1 Introduction . . . 15
3.2 Adaptation . . . 15
3.2.1 Basic Concepts . . . 15
3.2.2 Reasons to Adapt . . . 16
3.2.3 Types of Adaptive Services . . . 16
3.2.4 Adaptation Solutions . . . 18
3.3 How to Implement Adaptation . . . 20
3.3.1 Cloud Resource Management System . . . 21
3.3.2 State Machine Replication . . . 22
3.4 A Proposal for a CoC Adaptation Manager . . . 23
3.4.1 Overview . . . 23
3.4.2 Manager Internals . . . 25
3.4.3 Operations Required . . . 25
3.4.4 Manager Guarantees . . . 26
3.5 Related Work . . . 27
3.6 Conclusions . . . 27

**4 Object Storage: Revised DEPSKY Protocols and Evaluation** **29**
4.1 Introduction . . . 29
4.2 Cloud Storage Applications . . . 31
4.3 The DEPSKY System . . . 32
4.3.1 DEPSKY Architecture . . . 32
4.3.2 Data Model . . . 33
4.3.3 System Model . . . 34
4.3.4 Protocol Design Rationale . . . 35

4.3.5 DepSky-A– Available DepSky . . . 37
4.3.6 DepSky-CA– Confidential & Available DepSky . . . 38
4.3.7 Optimizations . . . 39
4.4 DepSky Extensions . . . 41
4.4.1 Supporting Multiple Writers – Locking with Storage Clouds . . . 41
4.4.2 Management Operations . . . 42
4.5 Cloud-of-Clouds Access Control . . . 44
4.6 Consistency Proportionality . . . 44
4.7 DepSky Implementation . . . 45
4.8 Evaluation . . . 46
4.8.1 Monetary cost evaluation . . . 46
4.8.2 Performance evaluation . . . 48
4.9 Related Work . . . 52
4.10 Conclusion . . . 54

**5 Object Storage: Theory 55**
5.1 Overview . . . 55
5.1.1 Motivation . . . 55
5.1.2 Contribution . . . 57
5.2 Model . . . 58
5.2.1 Executions . . . 58
5.2.2 Linearizability . . . 58
5.2.3 Wait-freedom . . . 59
5.2.4 Register Specifications . . . 59
5.2.5 Key-Value Store . . . 60
5.2.6 System model . . . 60
5.2.7 Consensus number . . . 60
5.3 On Robust Data Sharing with Key-Value Stores . . . 62
5.3.1 Algorithm . . . 62
5.3.2 Correctness . . . 65
5.3.3 Efficiency . . . 65
5.4 On Limitations of Using Cloud Storage for Data Replication . . . 67
5.4.1 The consensus number of a key-value store . . . 67
5.5 Related Work . . . 68
5.6 Conclusion . . . 69

**6 State Machine Replication: The Mod-SMaRt BFT Replication Protocol 71**
6.1 Introduction . . . 71
6.2 System Model . . . 73
6.3 State Machine Replication . . . 73
6.4 Validated and Provable Consensus . . . 74
6.4.1 Implementation requirements . . . 75
6.5 The Mod-SMaRt Algorithm . . . 75
6.5.1 Overview . . . 76
6.5.2 Client Operation . . . 77
6.5.3 Normal Phase . . . 77
6.5.4 Synchronization Phase . . . 79
6.5.5 Reasoning about the Consensus Modifications . . . 81

6.6 Optimizations . . . 81
6.7 Related Work . . . 83
6.8 Conclusion . . . 84

**7 Extensible ZooKeeper 86**
7.1 Introduction . . . 86
7.2 Background . . . 87
7.2.1 Coordination Services . . . 87
7.2.2 Usage Example: Priority Queue . . . 88
7.3 Enhancing Coordination . . . 88
7.3.1 Basic Approach . . . 89
7.3.2 Usage Example: Enhanced Priority Queue . . . 90
7.4 Extendable ZooKeeper . . . 90
7.4.1 Overview . . . 90
7.4.2 Managing an Extension . . . 91
7.4.3 Atomic Execution of an Extension . . . 92
7.5 Case Studies . . . 92
7.5.1 Priority Queue . . . 93
7.5.2 Quota Enforcement Service . . . 93
7.6 Related Work . . . 95
7.7 Conclusion . . . 97

**A Additional Material for Chapter 4 112**
A.1 Auxiliary functions . . . 112
A.2 Storage Protocols Correctness . . . 112
A.3 Lock Protocol Correctness . . . 114
A.4 Consistency Proportionality . . . 115

**B Proofs for Chapter 5 117**
B.1 Correctness of the Register Constructions . . . 117
B.1.1 MRMW-Regular Register . . . 117
B.1.2 Atomic Register . . . 120
B.2 Analysis of the Efficiency of Register Constructions . . . 121
B.3 Analysis of Cloud Storage Limitations . . . 123

**C Correctness of MOD-SMART 127**

# List of Figures

1.1 Graphical structure of WP2.2 and relations to other workpackages. . . . . . . . 5

2.1 Achieving cloud resilience with the TClouds cloud-of-clouds infrastructure . . 7
2.2 Diverse TClouds ecosystem . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.3 TClouds middleware . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.4 TIS - TClouds Information Switch . . . . . . . . . . . . . . . . . . . . . . . . 11
2.5 Byzantine-resilient protocols . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.6 Server-set implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

3.1 Types of adaptive services. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.2 Cloud-of-clouds adaptation manager architecture. . . . . . . . . . . . . . . . . 24
3.3 Software layers of the CoC adaptation manager. . . . . . . . . . . . . . . . . . 25

4.1 Architecture of DEPSKY (with 4 clouds and 2 clients). . . . . . . . . . . . . . 33
4.2 DEPSKY data unit and the 3 abstraction levels. . . . . . . . . . . . . . . . . . 34
4.3 DEPSKY read and write protocols. . . . . . . . . . . . . . . . . . . . . . . . . 35
4.4 The combination of symmetric encryption, secret sharing and erasure codes in DEPSKY-CA. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.5 Storage costs of a 1Mb data unit for different numbers of stored versions in different DEPSKY setups and clouds. . . . . . . . . . . . . . . . . . . . . . . 47
4.6 Latency of DEPSKY reads on real clouds. . . . . . . . . . . . . . . . . . . . . 49
4.7 Latency of DEPSKY writes on real clouds. . . . . . . . . . . . . . . . . . . . . 50

6.1 Modular BFT state machine replication message pattern. . . . . . . . . . . . . 72
6.2 MOD-SMART replica architecture. . . . . . . . . . . . . . . . . . . . . . . . . 76
6.3 Communication pattern of MOD-SMART normal phase for $f = 1$. . . . . . . . . 78
6.4 Communication steps of synchronization phase for $f = 1$. . . . . . . . . . . . 80

7.1 Callback mechanism usage example: An application process $p_1$ registers a data watch on a node; when the node's data is updated, the coordination service notifies $p_1$ about the modification. . . . . . . . . . . . . . . . . . . . . . . . 88
7.2 Pseudo-code implementation of a priority-queue client (ZooKeeper): an element is represented by a node, the priority is encoded in the node name. . . . . 89
7.3 Pseudo-code implementation of a priority queue in our extendable coordination service: the extension is represented by a virtual node `/queue`. . . . . . . . . 91
7.4 Throughput (i. e., successful dequeue operations) for different priority-queue implementations for different numbers of consumer processes. . . . . . . . . . 92
7.5 Pseudo-code implementation of a quota-server client in ZooKeeper: the current amount of free quota is stored in the data of `/memory`; to release quota, `allocate` is called with a negative `amount`. . . . . . . . . . . . . . . . . . . 93
7.6 Pseudo-code implementation of a quota server in our extendable coordination service: a call to `setData` only aborts if there is not enough quota. . . . . . . 95

7.7 Throughput (i. e., successful allocation and release operations) for different quota-server variants; the total quota is limited to the demand of 15 clients. . . 96
7.8 Costs (i. e., remote calls per operation) for different quota-server variants; the total quota is limited to the demand of 15 clients. . . . . . . . . . . . . . . . . . 96

# List of Tables

3.1 Adaptation solutions related with requirements. . . . . . . . . . . . . . . . . . . . 19

4.1 Functions used in the DEPSKY-A protocols (implementation in Appendix A.1). 37
4.2 Functions used in the DEPSKY-CA protocols. . . . . . . . . . . . . . . . . . . . . 41
4.3 Monetary costs of operating DEPSKY in real clouds. . . . . . . . . . . . . . . . 46
4.4 Clouds accessed during DEPSKY reads on real clouds. . . . . . . . . . . . . . . 50
4.5 Throughput observed in kb/s on all reads and writes executed for the case of 4 clouds ($f = 1$). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
4.6 The perceived availability of the used clouds. . . . . . . . . . . . . . . . . . . . 52

6.1 Variables and functions used in Algorithms 11 and 12. . . . . . . . . . . . . . . 79

# Chapter 1

# Introduction

## 1.1 TClouds — Trustworthy Clouds

TClouds aims to develop *trustworthy* Internet-scale cloud services, providing computing, network, and storage resources over the Internet. Existing cloud computing services are today generally not trusted for running *critical infrastructure*, which may range from business-critical tasks of large companies to mission-critical tasks for the society as a whole. The latter includes water, electricity, fuel, and food supply chains. TClouds focuses on power grids and electricity management and on patient-centric health-care systems as its main applications.

The TClouds project identifies and addresses legal implications and business opportunities of using infrastructure clouds, assesses security, privacy, and resilience aspects of cloud computing and contributes to building a regulatory framework enabling resilient and privacy-enhanced cloud infrastructure.

The main body of work in TClouds defines a reference architecture and prototype systems for securing infrastructure clouds, by providing security enhancements that can be deployed on top of commodity infrastructure clouds (as a cloud-of-clouds) and by assessing the resilience, privacy, and security extensions of existing clouds.

Furthermore, TClouds provides resilient middleware for adaptive security using a cloud-of-clouds, which is not dependent on any single cloud provider. This feature of the TClouds platform will provide tolerance and adaptability to mitigate security incidents and unstable operating conditions for a range of applications running on a clouds-of-clouds.

## 1.2 Activity 2 — Trustworthy Internet-scale Computing Platform

Activity 2 carries out research and builds the actual TClouds platform, which delivers trustworthy resilient cloud-computing services. The TClouds platform contains trustworthy cloud components that operate inside the infrastructure of a cloud provider; this goal is specifically addressed by WP2.1. The purpose of the components developed for the infrastructure is to achieve higher security and better resilience than current cloud computing services may provide.

The TClouds platform also links cloud services from multiple providers together, specifically in WP2.2, in order to realize a comprehensive service that is more resilient and gains higher security than what can ever be achieved by consuming the service of an individual cloud provider alone. The approach involves simultaneous access to resources of multiple commodity clouds, introduction of resilient cloud service mediators that act as added-value cloud providers, and client-side strategies to construct a resilient service from such a cloud-of-clouds.

WP2.3 introduces the definition of languages and models for the formalization of user- and

application-level security requirements, involving the development of management operations for security-critical components, such as “trust anchors” based on trusted computing technology (e.g., TPM hardware), and it exploits automated analysis of deployed cloud infrastructures with respect to high-level security requirements.

Furthermore, Activity 2 will provide an integrated prototype implementation of the trustworthy cloud architecture that forms the basis for the application scenarios of Activity 3. Formulation and development of an integrated platform is the subject of WP2.4.

These generic objectives of A2 can be broken down to technical requirements and designs for trustworthy cloud-computing components (e.g., virtual machines, storage components, network services) and to novel security and resilience mechanisms and protocols, which realize trustworthy and privacy-aware cloud-of-clouds services. They are described in the deliverables of WP2.1–WP2.3, and WP2.4 describes the implementation of an integrated platform.

## 1.3 Workpackage 2.2 — Cloud of Clouds Middleware for Adaptive Resilience

The overall objective of WP2.2 is to investigate and define a resilient (i.e., secure and dependable) middleware that provides an adaptable suite of protocols appropriate for a range of applications running on clouds-of-clouds. The key idea of this work package is to exploit the availability of several cloud offers from different providers for similar services to build resilient applications that make use of such cloud-of-clouds, avoiding the dependency of a single provider and ruling out the existence of Internet-scale single point of failures for cloud-based applications and services.

During the first year, a set of components and algorithms were identified together with a reference architecture and described in D2.2.1. In this second year we focus on subset of these components, that were selected among the ones firstly proposed to be further developed and demonstrated at the end of year 2, and later exploited in the use cases defined in A3.

## 1.4 Deliverable 2.2.2 — Preliminary Specification of Services and Protocols of Middleware for Adaptive Resilience

**Overview.** The increasing maturity of cloud computing technology is leading many organizations to migrate their IT solutions and/or infrastructures to operate completely or partially in the cloud. Even governments and companies that maintain critical infrastructures are considering the use of cloud offers to reduce their operational costs [Gre10]. Nevertheless, cloud computing has limitations related to security and privacy, which should be accounted for, especially in the context of critical applications.

These limitations are mostly related with the fact that currently cloud-based applications heavily depend on the trustworthiness of the cloud provider which is offering the resources or services. However, the trust put on the cloud providers may be unjustified due to a set of threats that may affect the services:

- **Loss of availability**: When data is moved from the company’s network to an external datacenter, it is inevitable that service availability is affected by problems in the Internet. Unavailability can also be caused by cloud outages, from which there are many reports

[Rap11], or by DDoS (Distributed Denial of Service) attacks [Met09]. Unavailability may be a severe problems for critical applications such as smart lighting.

- **Loss and corruption of data**: There are several cases of cloud services losing or corrupting customer data. Two elucidative examples are: in October 2009 a subsidiary of Microsoft, Danger Inc., lost the contacts, notes, photos, etc. of a large number of users of the Sidekick service [Sar09]; in February of the same year Ma.gnolia lost half a terabyte of data that it never managed to recover [Nao09].

- **Loss of privacy**: The cloud provider has access to both the stored data and how it is accessed. Usually it is reasonable to assume the provider as a company is trustworthy, but malicious insiders are a wide-spread security problem [HDS+11]. This is an especial concern in applications that involve keeping private data like health records.

- **Vendor lock-in**: There is currently some concern that a few cloud computing providers may become dominant, the so called vendor lock-in issue [ALPW10]. This concern is specially prevalent in Europe, as the most conspicuous providers are not in the region. Even moving from one provider to another one may be expensive because the cost of cloud usage has a component proportional to the amount of data that is transferred.

In this deliverable we further develop a model in which a set of cloud providers are used to build services such as object storage (in the same line as object-based cloud storage IaaS) and replicated execution (something similar to a PaaS). Since most contributions here presented revolve around the idea of replicating computation or information on several independent clouds, the bulk of the contribution in this deliverable comes in the form of a set of practical replication & coordination protocols, their evaluation, and the study of their theoretical limitations. All these contributions are related to a subset of components/ideas provided in the first year. In particular, we focus on three main components: cloud-of-clouds object storage, BFT asynchronous state machine replication and fault-tolerant BPEL and its efficient implementation. Moreover, in this deliverable we start discussing how these services can be made adaptive, with particular emphasis on adding and removing replicas for scale-up (how to exchange replicas by others more powerful) and scale-out (how to add more replicas to a system).

**Structure.** This deliverable is organized in the following way. Chapter 2 offers a concise description of the main architectural aspects of the revised TClouds reference architecture (initially described in D2.2.1, and latter published in [VBP12]). The next chapter (3) discusses the need for cloud services adaptation and proposes a framework for building adaptable dependable cloud services. We expect to fully develop and support this model in the third year of the project.

The next two chapters describe our contributions regarding the provision of a cloud-of-clouds object store in two complementary ways. Chapter 4 presents DEPSKY, a new storage protocol that integrates a set of techniques for Byzantine fault tolerance, cryptography and coding theory for providing reliable, secure and cost-effective storage using a set of storage cloud providers (e.g., Amazon S3, Windows Azure Blob, Google Storage) as long as more than 2/3 of them are correct and available. A preliminary design of DEPSKY was described in D2.2.1, but in this deliverable we present the consolidated protocols (updating also the published version [BCQ+11]) and a complete evaluation of the system using a diverse set of real cloud providers. The DEPSKY chapter is complemented by Chapter 5, where we discuss the theoretical framework for these object storage systems (here called key-value stores) and investigate

several issues and limitations related with their use and implementation. The material on this chapter is based on two published works [CJS12, BCE+12]. These two chapters provide practical (including deployment experience) and theoretical underpinnings about cloud-of-clouds object-based storage services, a concept introduced by TClouds.

In Chapter 6 we turn back to BFT state machine replication protocols and, in particular, the BFT-SMaRt replication library [LaS10], which has its architecture and initial implementation described in last year WP2.2 deliverable (D2.2.1). In this chapter we describe the theoretical foundations of the protocol used in BFT-SMaRt to order messages and ensure strong consistency for replicated services. The contributions presented in this chapter were published in [SB12].

Finally, Chapter 7 returns to the dependable web services orchestration framework defined in D2.2.1 (dubbed FT-BPEL). This chapter delves into the implementation of a component that is fundamental for this service, an extensible coordination service that can also be interesting to other services and applications.

At the end of the deliverable we provide an appendix containing proofs for several results presented in Chapters 4, 5 and 6.

**Deviation from Workplan.** This deliverable conforms to the DoW/Annex I, Version 2.

**Target Audience.** This deliverable aims at researchers and developers of security and management systems for cloud-computing platforms. The deliverable assumes graduate-level background knowledge in computer science technology, specifically, in operating system concepts and distributed systems models and technologies. Despite the effort of the authors to make the their contributions accessible for a broad range of readers, there are chapters such as 5 and 6 that will be better appreciated by readers with some background on distributed systems theory.

**Relation to Other Deliverables.** Figure 1.1 illustrates WP2.2 and its relations to other workpackages according to the DoW/Annex I (specifically, this figure reflects the structure after the change of WP2.2 made for Annex I, Version 2).

The present deliverable, D2.2.2, presents a set of contributions related with the use of multiple clouds to implement trustworthy services and applications. This deliverable refines the contributions described in last year deliverables (D2.2.1 and D2.1.1) and defines a subset of the components initially proposed to be demonstrated in the TClouds 2nd-year demo (mainly cloud-of-clouds object-based storage and BFT state machine replication), as described in D2.4.2. Naturally, most of the ideas presented here will be used in the components used in both use cases of A3.

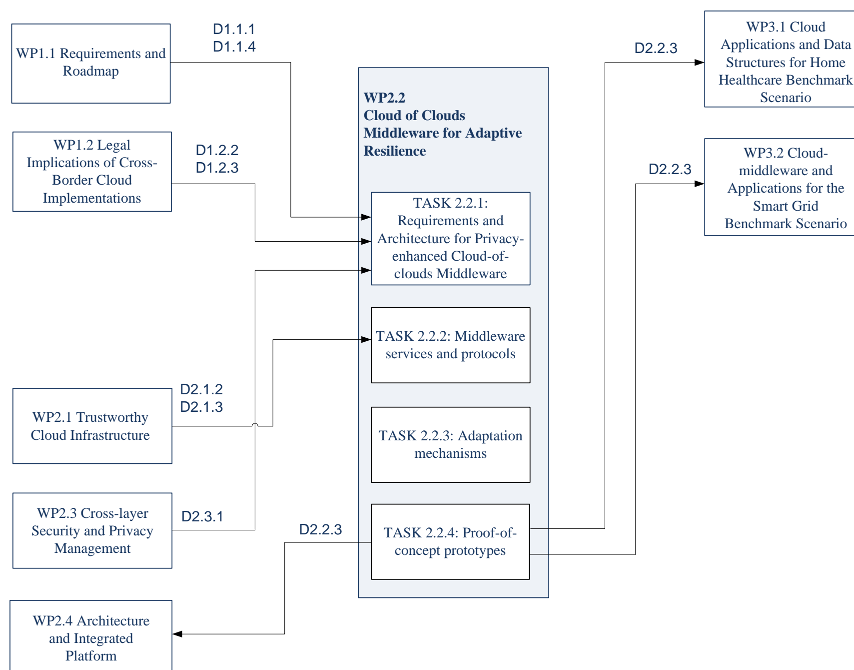



Figure 1.1: Graphical structure of WP2.2 and relations to other workpackages.

# Chapter 2

# Revised TClouds Architecture

*Chapter Authors:*
Paulo Veríssimo, Alysson Bessani and Marcelo Pasin (FFCUL).

## 2.1 Introduction

Cloud Computing (CC) has gained enormous momentum and is progressively reaching every sector using IT. However, as more and more services migrate to CC, so increases the dependence of the IT business on latter, perhaps not yet met by adequate levels of robustness. However, short of promising adequate security management of the infrastructure and perhaps some form of disaster recovery, little more has been offered by cloud providers so far. This can be testified by the numerous failures of cloud provider services made public, having caused service and data loss, as well as confidentiality compromises [BPN+11, Met09, Nao09, Sar09].

How is this being addressed today? Approaches confined to single cloud provision will not address high-resilience objectives, since they are a single point of failure. Large cloud providers attempt to achieve resilience by deploying several, differently located, cloud subsystems, which definitely improves on the situation, but they still remain under a single management and trust domain, with regard to common-mode and malicious faults. Federated cloud environments [RBL+09] certainly introduce the next instance of a solution, but they require alliance of the involved providers, and the general reality is that providers usually compete and may be mutually distrusting or even hostile to each other.

For the above reasons, we believe that built-in, open, and diverse solutions to cloud dependability and security are required. We propose the *cloud-of-clouds* paradigm as a path to achieve cloud computing resilience. The cloud-of-clouds paradigm extends the cloud concept, by leveraging the availability of multiple or federated cloud environments to create diverse ecosystems, and by letting users (besides providers) self-organize the way they use multiple cloud computing offerings.

Our intuition is that a resilient cloud computing infrastructure should (1) be based on a cloud-of-clouds setting; (2) achieve resilience against both attacks and accidents; (3) do so in as automated as possible a way; and (4) be open but not replace commodity clouds, instead, act in complement or in addition to them.

In this chapter we give a preliminary overview of the key points of the TClouds cloud-of-clouds reference architecture, which is devoted to providing incrementally high levels of security and dependability to cloud infrastructures, in an open, modular and versatile way. We describe initial validation of the architecture through recently published proof-of-concept systems and prototypes developed in the context of the project, which give promising prospects for this approach. This chapter is a consolidated and concise version of the initial architecture specification presented in 1st year WP2.2 deliverable (Chapter 4 of D2.2.1).

## 2.2 Motivation and Approach

In this section, we discuss the motivation and the approach to the model and architecture of TClouds. The diagnosis of the security and dependability problems faced by current cloud computing systems, which led to the design of TClouds, can be shortly described as follows:

- A cloud scenario has dependability and security needs that cannot be met by the application layer alone, requiring security-specific solutions to be provided at *lower layers* of the cloud architecture (infrastructure and platform).
- However, specific and proprietary IaaS or PaaS approaches to achieving resilience can make migration or interoperation difficult and expensive, creating vendor lock-in and competition exclusion [ALPW10], requiring instead *open* mechanisms.
- Finally, high-resilience objectives may be compromised, even for multiple or federated clouds, in solutions that may possess, at organizational level, single points of failure (e.g., common or related management and trust domains), requiring a genuinely diverse *cloud-of-clouds* approach.

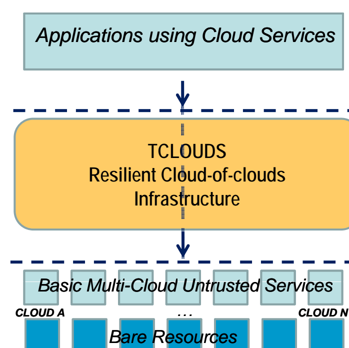



Figure 2.1: Achieving cloud resilience with the TClouds cloud-of-clouds infrastructure

Problems such as described above can be solved, and we propose to address them with the TClouds architecture, pretty much in the way depicted in Figure 2.1. In short, the architecture foresees the introduction of an infrastructure providing resilience, between commodity untrusted services, and the applications requiring security and dependability. This architecture aims at providing automated computing resilience against attacks and accidents in complement or in addition to classical commodity clouds protection schemes. It should do so whilst preserving the ability to freely choose from and use multiple cloud offerings without any pre-defined coordination amongst the cloud providers. This is the main difference to current federated cloud approaches. Furthermore, solutions should provide incremental levels of resilience, namely by modularly offering Iaas and PaaS level mechanisms.

In the remainder of the section, we discuss the main points behind the definition of the architecture.

**Promote automatic protection and avoid single points of failure** Clouds, being high-value, complex, large-scale and distributed infrastructures, are currently enduring high levels of threat. In order to achieve effective protection against such adverse conditions, solutions should involve

automatic mechanisms, such as those provided by intrusion tolerance [VNC03]. Algorithms and protocols based on this paradigm, such as Byzantine fault tolerance and proactive/reactive recovery [ADD+10, CL02, VCBL10] are addressed in TClouds. These techniques have been shown to successfully obtain high degrees of resilience in recent proof-of-concept prototypes for equally complex and exposed systems such as critical information infrastructures [BSC+08]. These mechanisms complement (not substitute) classical security techniques.

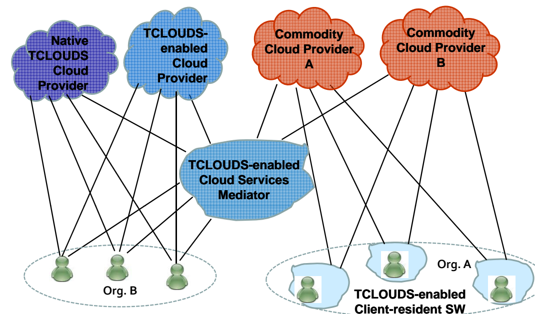



Figure 2.2: Diverse TClouds ecosystem

Intrusion-tolerant protocols, optimized to coordinate replicas, are fit to exploit the redundancy and diversity that comes from relying on multiple cloud providers, thus obviating the organizational syndrome of having single points of failure, mentioned earlier. The relevant mechanisms should, however, be as transparent as possible, offering cloud-of-clouds abstractions to the higher-level users, whilst hiding the complexity of managing that redundancy and diversity.

**Preserve legacy needs whilst enabling a diverse ecosystem** The architecture should accommodate multiple resilient cloud computing deployment alternatives, in order to be successful. It should ease migration of commodity cloud providers to whatever cloud resilience solutions to be advanced, by preserving legacy IaaS-level technology and components as much as possible. On the other hand, those solutions should be open, facilitating the emergence of new players such as intermediate added-value (e.g., resilient) cloud service providers, between the very-large-scale commodity cloud providers and the final end-users.

Avoiding an explosive growth of complexity, however, implies being modular and versatile enough to allow different instantiations under the same generic structure. As an example, it should support some key interaction modes: simultaneous use of commodity clouds from different providers (untrusted cloud-of-clouds); introduction of resilient cloud service mediators, acting as added-value cloud providers; accommodation of devices for in-house clouding (allowing an organisation to build its own resilient private cloud); support of lightweight end-users directly over commodity clouds; support of smooth migration paths for commodity providers toward TClouds-enabled resilient clouds.

Our vision for such a rich ecosystem is depicted in Figure 2.2. Technically, this vision translates to having the architecture be able to supply systems architects with all the alternatives above. We explain our approach to this key point later in the text.

The scenarios outlined in Figure 2.2 are a fairly complete combination of realistic resilient cloud computing deployment alternatives. For example, we foresee implementations of TClouds functionality preserving the use of legacy commodity clouds IaaS, either by resorting to TClouds-enabled client-side software deployed in-house, or to server-side software deployed

at an added-value trusted-clouds provider offering managed resilient cloud services. The architecture also allows for more ambitious steps, those considering that commodity cloud providers will eventually adhere to a model such as TClouds, directly providing resilient CC. In fact, our architecture foresees two basic paths for commodity cloud migration: (i) TClouds-enabled cloud, preserving data center machinery and software, and adding a wrapping resilience layer on top; (ii) TClouds native data centers and cloud, built from scratch with TClouds mechanisms.

## 2.3 Architecture Overview

Several mechanisms (replication, Byzantine fault-tolerance, proactive recovery, randomisation, trusted platform modules, etc.) are selectively used in the TClouds architecture, to build layers of progressively more trusted components and middleware subsystems (trusted IaaS and PaaS), from baseline untrusted components (basic multi-cloud untrusted services). This leads to an automation of the process of building trust: for example, at lower layers, basic intrusion tolerance mechanisms are used to construct a trustworthy communication subsystem, which can then be trusted by upper layers to securely communicate amongst participants, or to securely manage a set of replicas, without bothering about network or host intrusion threats.

The TClouds architecture builds on previous secure and dependable architectures, such as MAFTIA [VNC+06], for the lower level mechanisms, but extends them significantly to attend to the specific challenges of cloud computing infrastructures. As mentioned before, TClouds could be described, in short, as a *resilient cloud-of-clouds infrastructure* providing automated resilience against attacks and accidents, in complement or in addition to commodity clouds. This enhanced functionality is achieved through specialised *TClouds middleware* standing between low-level, basic multi-cloud untrusted services, and the applications requiring security and dependability, as depicted in Figure 2.3. This middleware is essentially of two classes: Trusted Infrastructure as a Service (T-IaaS) and Trusted Platform as a Service (T-PaaS).

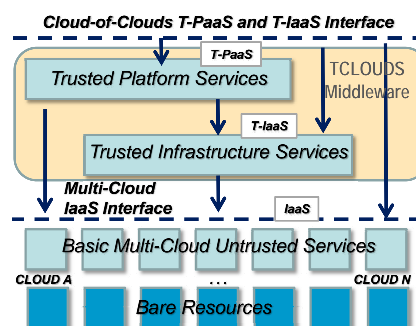



Figure 2.3: TClouds middleware

The characteristics of the architecture lead to the fulfillment of requirements outlined earlier. The TClouds middleware will be an enabler of these objectives, by providing:

- *Support for heterogeneity and openess.*
- *Transparent and incremental resilience with regard to failures of individual clouds.*
- *Modular functions and protocols.*

As suggested in Figure 2.3, the TClouds middleware lowest layer, Trusted Infrastructure as a Service (T-IaaS) provides services which are built on the services provided by the commodity untrusted clouds ecosystem, at the Multi-Cloud IaaS Interface, the lower interface of the TClouds middleware which, as the name says, provides the standard IaaS services, such as storage, processing, etc. The T-IaaS services can either be used directly by application users at the top TClouds middleware interface, the Cloud-of-Clouds Trusted Interface, or recursively by the next layer up, the Trusted Platform as a Service (T-PaaS). The latter services can also use basic cloud services, through the Multi-Cloud IaaS Interface.

Incremental resilience is achieved by selectively using the services provided at the Cloud-of-Clouds Trusted Interface, with different degrees of resilience, as exemplified in Figure 2.3: untrusted commodity cloud IaaS services; middleware T-PaaS, Trusted Platform Services; middleware T-IaaS, Trusted Infrastructure Services. All these services may also be provided by advanced TClouds implementations, over bare resources, by cloud providers (to sell resilient public clouds), or by other companies (to transform their IT into private clouds).

## 2.4 Main Building Blocks

The main building blocks of the architecture that implement the above-mentioned functionality, presented in Figure 2.3, are introduced and explained in this section.

The minimal functionality is offered by basic multi-cloud untrusted services. These blocks represent the currently standard functionality, at IaaS level, offered by commodity market players. Available services may evolve with the evolution of these systems, but are normally confined to storage, processing power, networking, and several input/outputs.

**Trusted Infrastructure and Platform Services** The Trusted Infrastructure Services building block represents trusted-trustworthy versions of IaaS services. The idea is to offer storage systems, and low-level virtual machines, resilient to attacks and faults, by combinations of fault-/intrusion prevention and tolerance mechanisms and protocols which build a resilience layer on top of the corresponding untrusted storage and processing systems.

The Trusted Platform Services building block represents trusted-trustworthy services at a higher level of abstraction, built on top of either or both the IaaS and the T-IaaS. These services normally support semantics useful to build complex reliable and distributed applications (for example, MapReduce). Once more, these services are implemented by combinations of fault-/intrusion prevention and tolerance mechanisms and protocols, for example, Byzantine fault-tolerant (BFT) protocols [ADD+10, VCBL10].

**TClouds Information Switches (TIS)** Let us recall our initial objective of allowing multiple TClouds deployments: final users, third-party added-value providers, or commodity providers wishing to directly offer some form of cloud resilience. To allow seamless deployment of the several alternatives, we need to materialize the middleware in a modular way that can morph to each particular configuration of TClouds. In order to do so, the resilience mechanisms are essentially based on a conceptual box, the *TClouds Information Switch* or TIS, a TClouds fundamental building block. The TIS runs the middleware protocols and mechanisms implementing the resilience components already mentioned (T-IaaS, T-PaaS) as shown in Figure 2.4.

The TIS is configurable, and each TIS instantiation encapsulates the services in use by that instance. As we see in the next section, these units constitute bricks which, with the adequate

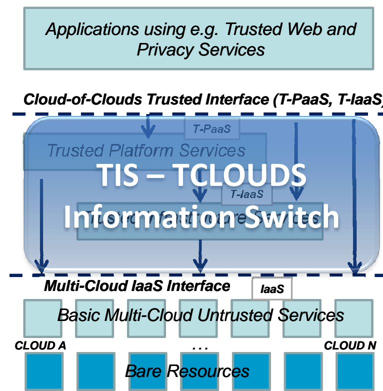



Figure 2.4: TIS - TClouds Information Switch

configuration and placement, materialize the several TClouds incarnations in a seamless and modular way.

As well as its configuration, the TIS implementation may also assume different forms, depending on the particular incarnation: dedicated machine; fault- and intrusion-tolerant appliance box containing several TIS replicas implemented as virtual machines. The TIS itself can be built with incremental levels of resilience against direct attacks, depending on its criticality (e.g., replicated hardware; replicated and diverse VMs on single hardware [RK07]; self-healing versions of the above [SBC+10]).

The TClouds Information Agent (TIA) is a software TIS, that allows configurable appliances residing with end clients, running essentially the same algorithms, but with minimal investment. On the other hand, the TIA option requires client modifications to achieve the desired TClouds functionality. Running in the client space, they are also subject to a greater level of threat, which can be mitigated by configurations where TIAs make use of trusted components.

## 2.5 Examples of TClouds distributed middleware

### Byzantine-resilient protocols

The workhorses of the solutions for achieving resilient cloud services are the so-called Byzantine fault-tolerant protocols, or BFT protocols. The baseline running environment for these modular protocols takes the form of software modules located with end-clients, e.g., to address the TClouds-enabled client-resident software alternative. As shown in Figure 2.5, these protocols may or may not involve communication between modules, besides direct communication with the several commodity clouds used.

In Chapter 4 we present a proof-of-concept prototype, concerned with resilient storage, DEPSKY (initial version published in [BCQ+11]): a resilient storage system that leverages the benefits of *cloud-of-clouds* architectures such as TClouds. DEPSKY relies on an efficient set of Byzantine quorum system protocols, cryptography, secret sharing, erasure codes and the diversity that comes from using several clouds.

Storage clouds are getting increasingly popular, but they have been affected by several problems, namely: loss of availability; loss and corruption of data; loss of privacy. The cloud-of-clouds paradigm is the crux of the DEPSKY approach to address these limitations.

Loss of availability or data is handled by using Byzantine fault-tolerant replication. *Byzantine quorum systems protocols* are actually used to store data on several clouds, overcoming

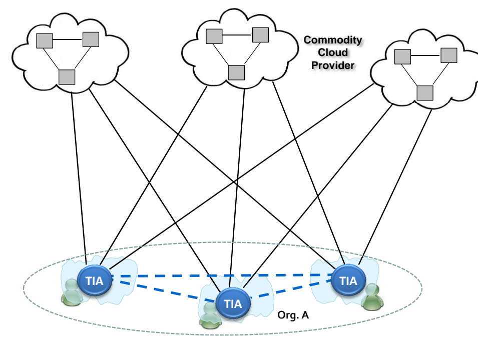



Figure 2.5: Byzantine-resilient protocols

single cloud failures. There is a risk of loss of privacy, because, technically, the cloud provider has access to data stored in the cloud, and as such so have malicious insiders or intruders. Encrypting the data before storing it is cumbersome for distributed applications with multiple users. DEPSKY employs a secret sharing scheme to avoid storing clear data in the clouds. Finally, erasure codes are used to improve the storage efficiency, amortizing the replication factor on the cost of the solution.

DEPSKY provides an extremely resilient solution depending on a 4-way cloud-of-clouds, however, at a price which is at most twice the cost of using a single cloud. This seems to be a reasonable cost, given the benefits. A set of real-life experiments were made, using four commercial cloud storage services (Amazon S3, Windows Azure, Nirvanix and Rackspace). Cost-performance ratio looks extremely promising for this kind of approach, as will be seen in Section 4.8.1.

DEPSKY is thus a virtual dependable and secure storage cloud, in reality developed across diverse commercial clouds. A user invoking the T-IaaS interface of a TClouds-enabled client-resident software module, such as depicted in Figure 2.5, would transparently obtain resilience of its storage, whilst invoking a regular storage service interface. On the other hand, the very same protocols might be easily adapted to run as a stand-alone storage service at a cloud services mediator, running on one or more servers of a server-set configuration as depicted in Figure 2.6, which in turn would interface with the required multiple commodity clouds. All details about the design and evaluation of DEPSKY will be presented in Chapter 4.

#### Server-set implementation

A powerful alternative is given by considering the implementation of similar protocols, by an actual set of distributed servers. The advantage is increased processing power and greater failure independence. With this configuration, it is technically feasible for a mediator, an intermediate provider, to establish the necessary IT to offer resilient cloud services to end-users, by buying untrusted commodity services and enhancing them with the TClouds middleware layer, as Figure 2.6 illustrates.

The upside is that client-side installations are no longer necessary and resilient cloud provision becomes totally transparent. The downside of a mediator may be that this brings us back to requiring trust in a single cloud provider, by the end users. This aspect can be mitigated by three arguments: (a) given that most of the resources and processing power still lie with the raw IaaS from the commodity providers, and not with the (relatively tiny) middleware in the mediator, most of the failure risk still lies with the former providers, and that is mitigated by the diversity of the untrusted multi-cloud ecosystem; (b) the resilience measures we foresee to implement the

TIS further help mitigate the risk of a single point of failure on the mediator side and constitute objective evidence of trustworthiness that can be used by an independent provider toward its clients; (c) this mediator may in the end be organizationally trusted, such as a subsidiary of the end users' organization group providing resilient cloud solutions. At least in the early phases of the creation of a trusted clouds ecosystem, we believe this to be an interesting technological alternative.

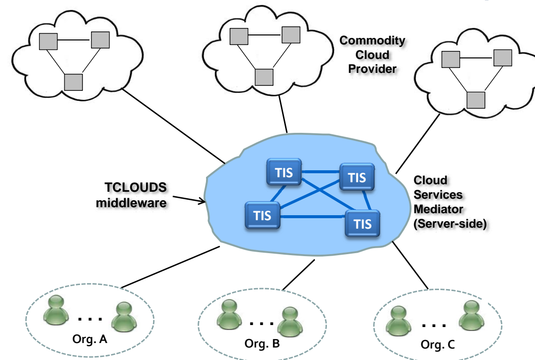



Figure 2.6: Server-set implementation

We have developed a second proof-of-concept prototype, around MapReduce. MapReduce is a framework developed by Google for processing large data sets [DG04], and its open counterpart, Apache Hadoop MapReduce, is largely used today, having become a typical PaaS-level service, deployed and used by many cloud computing companies (e.g., Amazon, EBay, Facebook, IBM, LinkedIn, RackSpace, Twitter, and Yahoo!). MapReduce runtimes like Hadoop tolerate crash faults, but not Byzantine faults. However, evidence in the literature shows that arbitrary faults do occur and can probably corrupt the results of MapReduce jobs [SG07].

In consequence, Hadoop seemed the perfect candidate to further experiment with the TClouds concepts, by considering a server set implementation of the architecture, accessing untrusted IaaS cloud services and implementing a trusted platform service, that is, a Byzantine fault-tolerant version of Hadoop. In the 1st year WP2.2 deliverable (Chapter 10 of D2.2.1, also published in [CPBC11]), we presented a MapReduce algorithm and prototype that tolerate these faults. This BFT MapReduce algorithm executes a job using only twice as many resources as the original Hadoop, instead of the 3 or 4 times more that would be required with the direct application of the usual quorum for Byzantine fault-tolerance.

### TIS-based TClouds-enabled cloud

In more advanced phases of the creation of a trusted clouds ecosystem, we believe that commodity cloud providers themselves will improve, even if selectively, their quality of service with regard to security and dependability. Competition from added-value providers will create a generic push for resilient solutions, of the kind offered by TClouds, which we discuss in this section and the next.

The first and easiest way for a commodity provider to offer resilient services is to preserve its investment in raw (non-resilient) cloud IaaS infrastructure, and implement the TClouds middleware on top of it. This will be implemented by several TIS topologically located in a way as to completely wrap the relevant cloud's data centers. Note that the TIS may be constructed to be as resilient as desired, as mentioned earlier.

### Native TClouds installation

The most advanced and effective solution is based on the implementation of a native TClouds system from scratch, over bare resources which can themselves already have local resilience mechanisms. TClouds WP2.1 is investigating architectural solutions for this scenario.

A TClouds native cloud is achieved by re-implementing the data centers to be TClouds compliant. This means at least two things: (a) the use of local fault/intrusion prevention and tolerance mechanisms to enhance the basic machines and resources, for example at the level of virtual machines and hypervisors, possibly relying on trusted components [CRS+11, KVB11]; (b) the re-design of the BFT protocols used to implement the TClouds middleware, to run embedded in the bare resources, making the latter intrinsically secure and dependable. That is, a native TClouds cloud will offer, from scratch, T-IaaS and T-PaaS, with the obvious gains in the trustworthiness–performance product and, possibly, in functionality.

## 2.6 Conclusion

We have presented a preliminary overview of the TClouds architecture, explaining our approach to provide incrementally high levels of security and dependability for cloud computing applications. We have shown that, through modularity and encapsulation around the TClouds Information Switches, the architecture's fundamental building block, it is straightforward to support a diversity of resilient cloud computing deployment alternatives. We exemplified some concrete instantiations of the architecture, which we believe validate the architecture's effectiveness in promoting open, modular and versatile resilient cloud computing.

# Chapter 3

# Adaptation

*Chapter Authors:*
Vinícius Cogo, Marcelo Pasin and Alysson Bessani (FFCUL).

## 3.1 Introduction

Elasticity (the capacity of a service grow and shrink as it workloads demand) is one of the five essential characteristics that define cloud computing [MG11]. The provision of this characteristic is strictly related to the adaptability of a cloud-based service and to the amount of resources allocated to it. Although elasticity is the most well-know adaptation property related to a cloud-based service, it is not the only dynamic adaptation available in the cloud. This chapter goes further and describes several possible solutions of dynamic adaptation in cloud environments, considering also the cloud-of-clouds scenario.

The chapter is organized as follows. First (Section 3.2) we review the basic concepts of adaptation, pointing reasons to adapt services and describe which type of services can be adapted in cloud environments. Then we discuss the requirements that some critical components, namely the resource manager and replication library, need to satisfy in order to implement adaptation (Section 3.3). After that, we propose an adaptation manager to perform dynamic adaptations in services running in a cloud-of-clouds scenario (Section 3.4). Finally, we describe some related work and final remarks (Sections 3.5 and 3.6).

## 3.2 Adaptation

### 3.2.1 Basic Concepts

Adaptation is the process of modifying the structure or behaviour of a service [Inv07]. A *static* adaptation means that possible changes are set before the service deployment or changes require some level of redeployment to be performed [FC09]. *Dynamic* adaptations may occur after the service deployment, meaning that the system does not need to be restarted [FC09].

Adaptations can be performed *manually* or *automatically*, where the latter consists in collecting data, evaluating it, making a decision and performing the changes. Adaptations can be either *reactive* or *proactive* [SBC+10]. The former is similar to an event-based approach, where events are collected and analysed to make decisions. The latter resembles a time-based approach, where changes are triggered independently, however the decisions still can be influenced by collected events.

Adaptations can be divided also in *parametric* or *compositional* [MSKC04, FC09]. The former consists in changing values of system parameters, which allows modifications only in

foreseen properties. The latter consists in exchanging algorithms and structural parts of a system, allowing initially unforeseen modifications.

### 3.2.2 Reasons to Adapt

Adaptation is employed to meet either functional or non-functional requirements in software engineering [Inv07]. From a service provisioning perspective, it is employed to meet requirements from service owners, users and environment. The former can be subdivided into inter-individual - different individuals - and intra-individual differences - individual evolution or behavioural changes. The latter consists mostly in environmental changes.

Performance, economy, security and legal reasons are the main causes to adapt services. Most quality of service (QoS) metrics are taken into account in service adjustments based on performance, for instance, an expected service throughput or latency. By contrast, most quality of protection (QoP) factors are important for adaptations based on security terms, for instance, the amount of faults that can be tolerated in a specific service. Budget reduction is the main reason for adaptations based on economic constraints. Additionally, issues related to local legislations, software licensing and copyright are the main reasons for adjustments based on legal terms.

### 3.2.3 Types of Adaptive Services

All type of services can be adapted, from the non-replicated service to the Byzantine fault-tolerant service based on state machine replication. This section describes the main types of adaptive services considering a partially synchronous model [DLS88], which are presented in Figure 3.1.

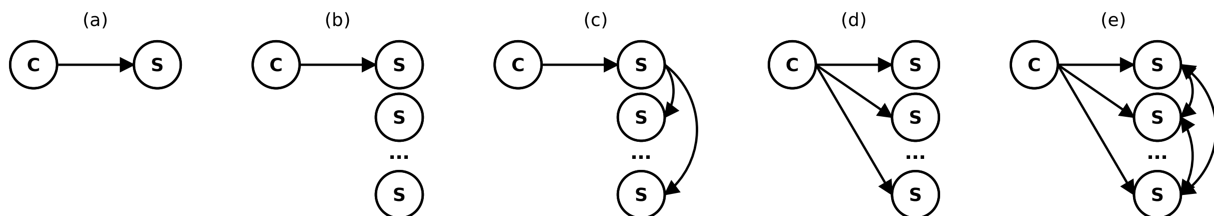



Figure 3.1: Types of adaptive services.

The types of adaptive services presented in Figure 3.1 are described bellow.

**(a) Non-replicated service.** A non-replicated service is the simplest type of adaptive services addressed in this work. It is composed of only 1 service instance that is responsible to process all client requests. It does not use any kind of replication to obtain fault tolerance or scalability.

Changing the amount of resources allocated to the service instance can adapt the service capacity to the desired level of QoS. *Live migration* is the process of moving a service instance from one physical host to another within the same cloud or even to a different cloud provider. It is the main action performed to adapt this kind of service, but it can cause small disruptions on service provisioning.

**(b) Stateless crash fault-tolerant services.** This is the first type of service considering replication to obtain fault tolerance. It is composed of $n$ service instances, where at least one of them should process and answer client requests. Due to the stateless property, client requests are processed independently from all other requests. This type of service has a local state but it is not durable, which means that if a service replica crashes, the entire service state remains intact in a shared storage system.

Clients can send their requests to only one instance per time or can request to all instances at the same time. In the former case, crash faults are tolerated sending the same request to another replica, while in the latter clients must wait only the fastest reply. To tolerate $f$ crash faults, this type of service must consider deploying at least $f + 1$ service instances.

Adaptations in all replicated services can modify the number of service instances and their capacity. Changing the number of replicas can modify the QoP of replicated services, since the more instances you have, the more crash faults can be tolerated. Changing the amount of resources allocated to a service instance can adapt the service capacity to the desired level of QoS, in the same way, adding more replicas improve the capacity of service provisioning. Live migration, *replica replacement* and *inserting/removing instances* are the main actions performed to adapt this type of service.

**(c) Crash fault-tolerant services based on state machine replication.** This type of service is similar to the previous one, once it is also composed of $n$ service instances, where at least one of them should process and answer client requests. The main difference in this case is the state machine approach, where processing each client request can both depend on the previous requests and on the current service state.

Clients normally send their requests to a specific service instance that is considered the primary replica. The primary replica receives the client request and forward it to all other secondary replicas, ensuring they process it in the same order. All service instances start processing from the same service state and arrive to a new common state, but only the primary replica reply the client requests. Crash faults on primary replica can be tolerated by sending the same request to any other secondary replica, because all of them have the same service state. To tolerate $f$ crash faults, this type of service must consider deploying at least $f + 1$ service instances.

Changing the amount of resources allocated to a service instance can adapt the service capacity to the desired level of QoS. Notice that adding more replicas does not improve the system capacity since all replicas have to execute all requests. Live migration, replica replacement and inserting/removing instances are the main actions performed to adapt this kind of service. Transferring the service state is required when new replicas join the group.

**(d) Stateless Byzantine fault-tolerant replicated services.** This is the first type of service considering intrusion tolerance [VNC03] in this work. It is composed of $n$ service replicas, where all of them must receive, process and reply client requests. Due to the stateless property, client requests are processed independently from all other requests.

Clients send their requests to all service instances at the same time and wait their replies. Clients have a voter component that decides the correct answer based on the majority of answers. Replicas do not need to communicate with each other, but arbitrary faults need to be detected and faulty replicas recovered. To tolerate $f$ arbitrary faults, this type of service must consider deploying at least $2f + 1$ service instances, to allow majority voting on the replies.

Changing the number of replicas can modify the service QoP, but in this case, two new replicas are needed to increment by one the number of faults to be tolerated. Changing the

amount of resources allocated to a service instance can adapt the service capacity to the desired level of QoS, in the same way, adding more replicas improve the client provisioning capacity of the system. Live migration, replica replacement and inserting/removing instances are the main actions performed to adapt this kind of service.

**(e) Byzantine fault-tolerant services based on state machine replication.** This is the last and most complex type of adaptive service considered in this work. It is similar to (c), once all replicas must also receive, process and reply client requests. The main difference in this case is the state machine approach, where all replicas must agree in a Byzantine consensus regarding the order of processing requests and each request can both depend on other previous requests and on current service state.

Clients send their requests to all service instances at the same time and wait their replies. Clients have a voter component that decides the correct answer based on the majority of answers. Replicas communicate with each other to achieve the Byzantine consensus, and arbitrary faults need to be detected and faulty replicas recovered. To tolerate $f$ arbitrary faults, this type of service must consider deploying at least $3f + 1$ service instances.

Changing the number of replicas can modify the service QoP, but in this case, three new replicas are needed to increment by one the number of faults to be tolerated. Changing the amount of resources allocated to a service instance can adapt the service capacity to the desired level of QoS, in the same way as (c), however adding replicas does not necessarily improve the capacity in this type of system. Live migration, replica replacement and inserting/removing instances are the main actions performed to adapt this kind of service. The main difference is that transferring the service state is required when new replicas join the group.

Notice that if more than $f$ faults happen in any type of replicated services (from *(b)* to *(e)*), then the fault or intrusion tolerance foreseen by this type of service will be compromised. Thus, the QoP provided by these services became the same as from the service *(a)*, which means that there is no protection from the replication mechanism.

There are other properties or factors that can be adapted in all foreseen types of adaptive services, not only the amount of service instances or their capacity. One example of property that can be adapted in all cases is the geographical location of service instances.

### 3.2.4 Adaptation Solutions

This section contains a discussion regarding the criteria and solutions to dynamically adapt services running in a multiple cloud environment, as considered in WP2.2. Table 3.1 correlates each solution with the correspondent criteria, which is described in the following.

Horizontal scalability is the ability of *(a)* **increasing** or *(b)* **reducing** the amount of computing instances responsible for providing a service. Increasing the number of service instances is a reactive action to deal with peaks of client requests. Additionally, the more service instances providing the service, the more faults can be tolerated, which leads to an improvement in the service's QoP. From a fault tolerance perspective, the more service instances providing the service, the more faults can be tolerated, which leads to an improvement in the service QoP.

Reducing the number of service instances can save resources and money. Economic constraints can lead to the necessity of reducing the number of service instances. Such reduction can also be performed by a proactive housekeeping process that periodically verifies if the amount of allocated resources is appropriate for the recent load.

Regarding the foreseen types of services, discussed in Section 3.2.3, scale out/in are employed more often to adapt stateless replicated services. Increasing the number of replicas of a

| | Performance | Economic | Security | Legal |
|---|---|---|---|---|
| *(a)* Increasing the number of service instances | X | | X | |
| *(b)* Reducing the number of service instances | | X | | |
| *(c)* Upgrading the resources of service instances | X | | | |
| *(d)* Downgrading the resources of service instances | | X | | |
| *(e)* Moving service instances to different cloud provider | X | X | X | X |
| *(f)* Moving service instances close to clients | X | | | |
| *(g)* Moving service instances away from attackers | X | | X | X |
| *(h)* Replacing faulty service instances | X | X | X | |
| *(i)* Replacing software | X | X | X | |
| *(j)* Software updates | X | X | X | |
| *(k)* Replacing old service instances | X | | X | |

Table 3.1: Adaptation solutions related with requirements.

BFT service based on state machine replication typically reduces the cost of read-only operations [Rei11], but increases the cost of write operations. Dynamically adapting the number of replicas may be used to increase service quality, but requires careful modifications to the BFT protocols in use [DK11].

*(c)* **Upgrade** and *(d)* **downgrade** are procedures to respectively increase and reduce the amount or capacity of resources allocated to service instances. Elasticity by means of vertical scalability is achieved through scaling up/down services. Upgrades can improve the service capacity, maintaining the amount of replicas. The more powerful the replicas are, the bigger is the quantity of requests per second processed by the service. Downgrades can release over-provisioned allocated resources, and consequently save money.

Parametric adaptation concerning CPU, memory and disk capacity are some examples of this adaptation. Usually, cloud providers have predefined categories of instances, called *tiers*, with predefined amount of resources. In this case, the parametric adaptation considers only category changes, for example, between small, medium, large and extra large instance configurations at Amazon Elastic Compute Cloud [ec2].

Considering the types of adaptive services discussed in Section 3.2.3, changing the capacity of service instances appear to fit well with services based on state machine replication, since it does not require modifying the BFT replication protocols. However, it can also be employed to adapt stateless services and non-replicated services.

*(e)* **Moving replicas to different cloud providers** can result in performance improvements due to different resource configurations, or financial gains due to different prices and policies on billing services. It is also important to prevent vendor lock-in, i.e., use more than one cloud provider to avoid an entire service or data set to be dependent on a specific provider, where changing it requires substantial costs [BCQ+11]. There are some initiatives to avoid or reduce costs in a vendor lock-in scenario, which encompasses a modular development of drivers for different clouds, usage of open interfaces, or usage of more than one provider since the beginning.[1] The more distributed among cloud providers a service is, the more protected against vendor lock-in it gets. Thus, a cloud-of-clouds adaptation manager must be able to allocate resources in multiple cloud providers, where they are not necessarily reliable. Additionally, different cloud providers can have distinct service policies. Moving the service to another provider can be favourable if the new service policy protects the service better than the old one.

Globally distributed services should take into account the logical or geographical location

[1] There are EU-funded projects addressing this problem, e.g., MOSAIC [mos]

of clients, thus *(f)* **moving service instances close to clients** can bring huge benefits in terms of performance. More specifically, achieving logical proximity to the clients can reduce the network latency between the service and clients. Moving service instances closer to clients may also place them within a different legislation, which can be favourable to the relationship between the service and its clients.

*(g)* **Moving replicas logically away from attackers** can increase the network latency experienced by the attacker, reducing the impact of its attacks on the number of requests processed (this is especially efficient for DoS attacks). Such intentional performance reduction can also reduce attacks effectiveness, e.g., brute force attacks. Geographical migrations of service instances can also place them somewhere with adequate laws against some types of attack.

*(h)* **Replacing faulty replicas** is a reactive process following a detection, which provides new correct instances in the place of faulty ones. A faulty replica can degrade the overall service performance, since there is one less instance providing it. The scenario can be even worse if the faulty replica becomes malicious and starts issuing expensive operations, such as unnecessary state transfers requests. The *pay-as-you-go* model bills service instances regardless if they are correct or faulty. Thus, if a faulty replica is not recovered, the service owner keeps paying for the resources allocated to that replica. Removing faulty replicas can also restore the fault tolerance guarantees of the service.

*(i)* **Software replacement** is an operation where software is replaced in all service instances at run-time. Operating systems, Web servers and database management systems are examples of software that can be replaced in this scenario. Different implementations might differ on performance aspects, licensing costs or security mechanisms.

The use of diverse operating systems in replicated services has good evidences of effectiveness on independence of faults [GBG+11]. Such diversity can also be obtained on each migration or replacement in order to make an attacker's life harder.

The *(j)* **software update** is the process of replacing software in all replicas by up-to-date versions. From a security perspective, vulnerable software must be replaced as soon as patches are available. New software versions might introduce optimized algorithms, thus improving the service performance.

The first steps to update a software consists in creating a new VM image with the new software version, registering it in at least one cloud and removing the old VM image from all cloud providers. The next steps come naturally from triggering adaptations in the cloud-of-clouds adaptation manager, which will select the new VM image in all adaptations.

Systems running for long time can be subject to ageing issues, where systems performance is degraded by long running effects. Software rejuvenation can be employed to *(k)* **avoid ageing problems**. It can be triggered by proactive process, which periodically verifies if there are old replicas running and replaces them if they exist.

Replacing old replicas can improve the system performance, because ageing issues can cause performance degradation. This adaptation can also avoid servers that contain non-patched vulnerabilities, or even recover replicas that are subject to ageing effects triggered by internal errors which had not been detected.

## 3.3 How to Implement Adaptation

This section contains a review of the requirements that a cloud resource management system and a replication middleware should provide in order to allow service adaptation in cloud computing. The first part is dedicated to describe existent scheduling solutions on cloud resource

managers, for instance, OpenStack and OpenNebula. The second part presents a context regarding reconfiguration in state machine replication and describe the protocol implemented in the BFT-SMaRt, a library for Byzantine fault-tolerant services based on state machine replication.

### 3.3.1 Cloud Resource Management System

Preparing the deployment of a new service instance is an important step for adapting services in the cloud. It encompasses choosing the ideal set of resources, which directly depends on the allocation features supported by cloud resource managers. Thus, the first requirement for cloud managers is a resource allocator considering more than one scheduling policy. A scheduling policy is an algorithm that selects resources from a snapshot of the current system state, considering some predefined criteria, as for example, the system load. OpenStack scheduler contains three predefined policies:

- Chance (random): Considering all available nodes that has enough resources to allocate the requested service instance, the scheduler choses one randomly.
- Simple: Select the least loaded host that has enough resources to allocate the requested VM.
- Zone: Choose a random host from all available hosts in a specified zone.

OpenNebula scheduler has also three predefined policies:

- Packing: Choose the node with more VMs running and with enough available resources to the requested service instance.
- Striping: Select the node with less VMs running and with enough available resources to the requested service instance.
- Load-aware: Choose the least CPU loaded host that has enough resources to allocate the requested VM.

Allowing users to create custom scheduling policies is the second requirement, since there are more criteria to be considered in placement policies than only the system load. OpenStack and OpenNebula schedulers allow users to create custom placement policies based on matchmaking algorithms [RLS98, AMM05]. This type of algorithm is composed of three steps: (1) filtering out the elements that are incapable of fulfilling the request, (2) ranking the remaining elements and (3) selecting the highest ranked element. There are several properties that can be used to compose a custom placement policy, e.g., the free, used or total amount of CPU, memory and hard disk, the virtualization environment, the geographic location and others. OpenStack and OpenNebula also allow users to create new custom properties to be considered in custom placement policies.

The third and last requirement for cloud managers is the provisioning of public APIs that allow clients to describe the resource requirements for new service instances. OpenStack provides a command line interface, a ReSTful HTTP service, called Compute API (compatible with the Amazon EC2 API), and a web-based interface, called Dashboard. OpenNebula provides a command line interface, two ReSTful HTTP services (one compatible with the Amazon EC2 API and other with OCCI [occ]), a XML-RPC interface and a web-based interface, called Sunstone.

### 3.3.2 State Machine Replication

In order to be used in adaptive systems a state machine replication (SMR) middleware must have the capacity of runtime reconfiguration. This means that the system must be able to add and remove replicas to a replica group implementing a service without stopping serving client requests. In this section we discuss the rationale for reconfiguring a SMR, list some related work and discuss the general approach we are implementing in the BFT-SMaRt replication library [LaS10] (described in Chapter 8 of D2.2.1).

**Context and related work.** A fundamental requirement of the state machine replication approach for fault tolerance [Sch90, CL02] is the coordination among replicas (see Section 3.1.2 of D2.2.1). This requirement can be translated in the need of running a consensus protocol among the group of service replicas to ensure that they process the same sequence of requests. This protocol can only run under a static group of processes in which every process know all other processes in the group[2].

The aforementioned limitation defines the problem of SMR reconfiguration in such a way that the reconfiguration commands, which change the replica group, must be processed between two consecutive consensus executions. This idea was first introduced in the seminal Paxos paper [Lam98], and more recently implemented in two crash fault-tolerant systems [LAB+06, SRMJ12]. However, there is still no solution for the reconfiguration of Byzantine fault-tolerant SMR.

**Reconfiguration on BFT-SMaRt.** We are currently working on a reconfiguration algorithm for the BFT-SMaRt replication library [LaS10]. In a scenario in which processes can be subject to Byzantine faults it is dangerous to let any party add or remove replicas, since malicious parties can change the system to add malicious processes or remove correct replicas to the replicas group. In both cases, these actions may cause the number of faulty replicas to be greater than $f < n/3$ (being $n$ the total number of replicas).

To avoid these problems, BFT-SMaRt assumes that only two kind of parties can invoke reconfiguration commands:

- Each replica can ask for its own removal from the group. The rationale is that the worst possible attack that can be made with this feature is a a faulty replica removing itself from the group, which is equivalent (or even better) than having a faulty replica in the system.
- There is a trusted third party called *reconfigurator* that is authorized to add and remove replicas from the group. The reconfiguration commands sent by this priviledged party are signed with a special private key, allowing replicas to verify received reconfiguration commands using the corresponding public key.

In our architecture there are just two reconfiguration commands to change the replica group of a service: ADD and REMOVE. These commands are processed in the following way.

1. The reconfiguration client (either a leaving replica or a reconfigurator) invokes ADD or REMOVE just as any other application-level service operation.

[2] There are theoretical solutions for the consensus problem with unknown participants (e.g., [ABFG08]), but these algorithms are still not efficient enough to be used in practice.

2. The service totally order the reconfiguration command using the Mod-SMaRt algorithm implemented in the system (see Chapter 6), i.e., the reconfiguration command is ordered through a consensus execution.
3. The replicas currently in the group process all requests in the batch[3] containing the reconfiguration request and, after it, process the group change.
4. The last step of the protocol depends on what happened with the replica during the reconfiguration:
   (a) The replicas staying in the system after the reconfiguration need to update their values of $n$ and $f$ and open (resp. close) connections with the arriving (resp. departing) replicas.
   (b) Departing replicas need to stay in the system until it sees $2f + 1$ replicas of the new view are processing requests on next agreement.
   (c) New replicas need to receive the state of the replication library (including $n$, $f$ and other configuration parameters) and the state of the service before starting processing further requests. This step is simplified by the reuse of the state transfer protocol implemented on BFT-SMaRt.

Currently, we are refining the implementation of this protocol and working on its formal description and proof. We expect to present a full description and experimental results of this work on year 3 WP2.2 deliverable (D2.2.3).

## 3.4 A Proposal for a CoC Adaptation Manager

This section presents an architecture for a cloud-of-clouds adaptation manager. We also describe some internal aspects of the adaptation manager, the list of operations identified as important to it and the foreseen guarantees and responsibilities.

### 3.4.1 Overview

Figure 3.2 contains the preliminary architecture of our CoC adaptation manager. From this point on, we consider that the service is completely deployed and running in order to perform the dynamic adaptations.

The *monitoring system* is the first component to be described, considering the top-down data flow sequence. It is typically composed of probes and sensors that collect data regarding users, components and environment of a service. Some examples of data that can be collected are: the service load, the uptime of service components, and other metrics important to monitor the system health. Such data can be stored in a persistent storage, processed by a decision engine, by a security information event manager (SIEM), or even presented to the service administrator through a dashboard.

*Decision engines* and *security information event managers* process and analyse the monitoring data, and trigger adaptation events. SIEMs are specifically responsible by events related with security, from access to the vulnerability management. Decision engines are responsible

[3] For performance reasons, the agreement on SMR is usually executed on a batch of messages to be ordered [CL02].

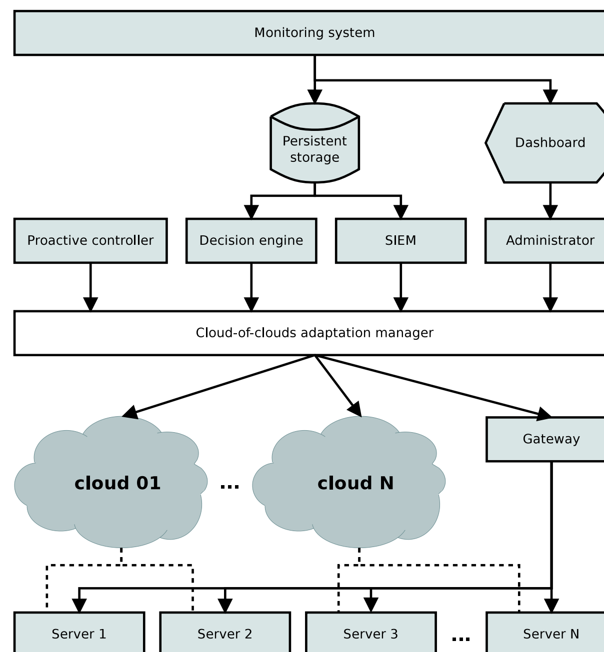



Figure 3.2: Cloud-of-clouds adaptation manager architecture.

by all events related with performance, economy and legal aspects of the service, including the enforcement of service level agreements (SLAs). Both can automatically produce requests to the adaptation manager, while the latter can also serve as a decision support engine to a service administrator.

The *administrator* role is important in cases where the automatic tools are not capable to measure all advantages, disadvantages and risk involving each service adaptation. There are cases where human intervention is imperative, mainly regarding the know-how of senior administrators, which can decide differently from adaptation algorithms based in business rules that are not completely implemented in automatic decision engines.

The last adaptation manager client is the *proactive controller*, which triggers dynamic adaptations based only on scheduling time. Proactive requests can be sent to the adaptation manager in the same way as in previous cases. Software rejuvenation is an example of proactive action regarding dynamic adaptation, where the oldest service instance is periodically replaced by a new and fresh one.

The *cloud-of-clouds adaptation manager* is responsible to perform dynamic adaptations in services following instructions and requests from proactive and reactive components. It must provide public interfaces that could be used by human administrators, automatic reactive decision engines, SIEMs and proactive processes. It must also be able to at least insert, remove, replace, grow or shrink service instances, considering multiple cloud providers and service protocols to join and leave replicas.

The *gateway* is the component responsible to maintain the group membership of each service and to provide this information to service clients. The adaptation manager has the responsibility to inform the service gateway about modifications in the service membership each time a change occurs. The gateway is a very simple component that works just like a lookup service, where new clients and instances request the most up-to-date group of replicas of a given service. It can also support pluggable functionalities, for instance, proxy and load balancing.

The service is composed of many *service instances* distributed in multiple cloud providers,

independently of being public, private or hybrid clouds. It can contain a protocol to join and leave service instances, as well as, it can use a gateway component as location service.

### 3.4.2 Manager Internals

Internally to the adaptation manager, it can be decomposed into many software layers that perform the dynamic adaptations, as show in Figure 3.3. The first one is an API layer that receives

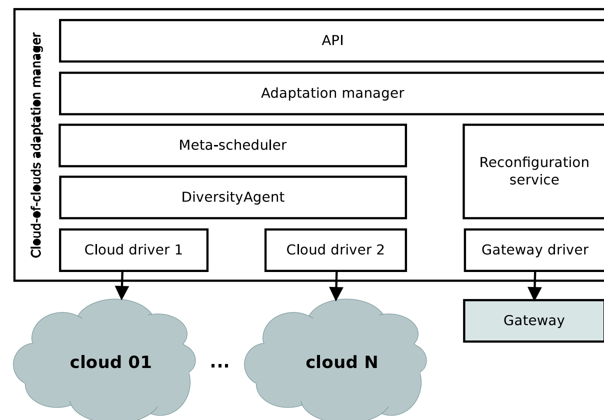



Figure 3.3: Software layers of the CoC adaptation manager.

the adaptation requests from the users of the CoC adaptation manager. A parser component can be responsible to parse the requests and forward correctly the internal requests to a service manager component.

The adaptation manager layer maintains information about the service in question and controls the gateway component, sending updates through the gateway driver. The meta-scheduler layer deals resources as a cloud broker, using multiple cloud providers in order to obtain the best resource combination to the adaptation requests. It can use the adaptation manager database or can locally maintain information regarding current and historical usage of cloud providers and their resources. The meta-scheduler can use a diversity component to request the most diverse resource combination considering the previously selected requirements. This can improve the independence of faults as discussed in Section 3.2.4.

The last layer encompasses the drivers, which are connectors between the manager and cloud providers or service gateways. These drivers are responsible to translate from an internal grammar to the grammar of the selected cloud provider.

### 3.4.3 Operations Required

In this section we describe the operations that a CoC adaptation manager should support.

**Migrating a service instance.** This operation moves a service instance to a new physical host, with minimal service downtime as possible. In most cases, this physical host belongs to the same cloud provider. Migrating a service instance to a physical host in a different cloud provider requires more time to conclude the process.

In stateless services, the migration could be done as a replacement, since replicas do not have a consistent or durable service state. In services that have a service state the migration must consider transferring the entire current state or image, like a snapshot.

There are security problems in migrating VMs that still need to be addressed. For example, the entire VM state is exposed during transmission, which means that OS/kernel memory, application state, sensitive data, passwords and keys will pass through the network. An unauthenticated and insecure migration plane is a concern, thus management capabilities should include authentication, confidentiality and isolation concerns [OCJ08].

The man-in-the-middle attack have to be avoided during a migration. This attack consists in placing a malicious component in the middle of the migration, between the two involved hosts. This component can act passively, sniffing sensitive data, passwords, keys in memory or actively, manipulating authentication services or kernel structures [OCJ08]. The solutions to this case encompasses using a separated and secure network for migration, using hardware-based cards like Trusted Platform Modules, maintaing the Virtual Machine Monitor or Hypervisor always up-to-date/patched, and encrypting the VM image before starting the migration (even it being really costly) [SRG12].

**Adding or removing a service instance.** These operations consist respectively, in deploying a new VM and inserting it to the group of service replicas, or removing an existent service instance from this group. It can use exactly the same VM image as other service instances have, or it can use a completely different VM image. In most diversities cases, a different VM image will be required.

New instances for stateless services only need to be registered in the gateway to make it able to start processing client requests. In stateful services, the service state must be transferred to the new replica, beyond the deployment and registration steps. Thus, after the state transfer finishes, the new service replica can start processing the client requests and achieve the same agreements as other replicas.

**Replacing a service instance.** A replacement encompasses inserting a new replica and removing and old one. The adaptation manager must deploy a new VM, insert it to the service group, remove the old replica and destroy the corresponding VM.

**Growing or shrinking a service instance.** These operations consist in modifying properties or the amount of resources allocated to a specific service instance. Nowadays, hypervisors provide only migration processes, which prohibit from growing or shrinking replicas that are active. Hypervisors should provide interfaces to increase or reduce the amount of resources allocated to a VM without causing a service disruption. This action could be performed similarly to a *realloc* for memory, also being possible for CPU and disk resources. Since there is this impossibility the adaptation manager must perform a migration operation in this case.

### 3.4.4 Manager Guarantees

**Intrusion detection.** The adaptation manager should guarantee that the new replica will not contain the same intrusion or fault as the removed instance. For that, the adaptation manager has to deploy a new instance based in a clean VM image.

**Load changes.** The adaptation manager cannot guarantee that an adaptation will solve the service requirements if the requested adaptation is not enough. If an adaptation is not enough to solve the load issues, service owners must follow monitoring and verifying that the adaptations done were not enough.

**Periodic triggers.** Triggering adaptations periodically is not an issue for the adaptation manager, because proactive controller must triggers the adaptations correctly on schedule. This proactive controller is the component that must guarantee that a new adaptation request is triggered on each $T$ time units.

**Replacement with deadlines.** Proactive recovery protocols consider strong time guarantees. The adaptation manager should be able to guarantee that a new VM will be ready to join the group before a specified deadline, or the replacement may fail. The same is considered for replicas removals, where the adaptation manager must guarantee that an instance is removed between a minimal and maximum timestamps.

## 3.5 Related Work

Several commercial providers support elastic adaptation in their clouds [amaa, raca, goG, ibm, win]. Most of their users take into account only performance aspects, adapting to load peaks. Some researches focus on employing elasticity based on the relation cost benefit of each adaptation [JHJ+10,SSSS11]. In this chapter we discussed about other solutions for service adaptation beyond the elasticity, for instance: service migration, replacement, recovery and rejuvenation.

We also presented other criteria to adapt further than performance and cost. We considered security as one important criteria to adapt, mainly related to the quality of protection (QoP) of a service.

A work that is related with ours is the one of a group of researchers from National ICT Australia [nic], which created a company called Yuruware [yur] for providing commercial adaptation solutions that goes further than pure elasticity. They provide globally distributed service replication, synchronization, monitoring, failure detection and recovery. They also provide a housekeeping process that periodically release idle resources, saving money for their clients. There are two important differences between the TClouds work and what they are providing. First, we support a more broad failure model (Byzantine) while they, in principle, are only concerned with crashes and unavailability. Second, they only use Amazon EC2 resources, which even having data centres distributed around the globe, is still a single point of failure in terms of administration. Our architecture, by the other hand, supports the use of multiple cloud providers.

## 3.6 Conclusions

This chapter focused in discussing service adaptation in cloud environments. We first reviewed the basic concepts of service adaptation and analysed the main reasons to adapt. Beyond establishing performance, economy, security and legal requirements as the main reasons for service adaptation, we presented a set of service types considered as adaptive. Such classification encompasses non-replicated services, crash or Byzantine replicated services, stateless or services based on state machine replication. Each adaptation criteria was correlated with service adaptation solutions, describing in which cases they can be employed.

We proposed a system component responsible for adapting services running in cloud environments, which we called the cloud-of-clouds adaptation manager. Such component is integrated with monitoring tools and decision engines that send adaptation requests accordingly to the service status, and with cloud resource managers that will receive requests from it to

allocate, deploy and release resources for the service in question. We also presented several requirements focusing cloud resource managers and state machine replication middleware. Four adaptation operations were identified as important for an adaptation manager implementation, including: migrate, add/remove, replace, grow/shrink service instances.

Our future works encompasses implementing the proposed CoC adaptation manager, including support for multiple drivers to deal with different cloud providers and integrate it with reactive and proactive controllers.

# Chapter 4

# Object Storage: Revised DEPSKY Protocols and Evaluation

*Chapter Authors:*
Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André and Paulo Sousa (FFCUL).

## 4.1 Introduction

The increasing maturity of cloud computing technology is leading many organizations to migrate their IT infrastructure and/or adapting their IT solutions to operate completely or partially in the cloud. Even governments and companies that maintain critical infrastructures (e.g., healthcare, telcos) are adopting cloud computing as a way of reducing costs [Gre10]. Nevertheless, cloud computing has limitations related to security and privacy, which should be accounted for, especially in the context of critical applications.

In this chapter we present the TClouds' first contribution towards the development of resilient cloud-based storage solutions. More specifically, we present DEPSKY, a dependable and secure storage system that leverages the benefits of cloud computing by using a combination of diverse commercial clouds to build a *cloud-of-clouds*. In other words, DEPSKY is a virtual storage cloud, which is accessed by its users to manage *updatable data items* through the invocation of operation in several individual clouds. More specifically, DEPSKY addresses four important limitations of cloud computing for data storage in the following way:

- **Loss of availability**: When data is moved from the company's network to an external datacenter, it is inevitable that service availability is affected by problems in the Internet. Unavailability can also be caused by cloud outages, from which there are many reports [Rap11], or by denial-of-service attacks like the one that allegedly affected a service hosted in Amazon EC2 in 2009 [Met09]. DEPSKY deals with this problem by exploiting replication and diversity to store the data on several clouds, thus allowing access to the data as long as a subset of them is reachable.

- **Loss and corruption of data**: there are several cases of cloud services losing or corrupting customer data. For example, in October 2009 a subsidiary of Microsoft, Danger Inc., lost the contacts, notes, photos, etc. of a large number of users of the Sidekick service [Sar09]. The data was recovered several days later, but the users of Ma.gnolia were not so lucky in February of the same year, when the company lost half a terabyte of data that it never managed to recover [Nao09]. DEPSKY deals with this problem using Byzantine fault-tolerant replication to store data on several cloud services, allowing data to be retrieved correctly even if some of the clouds corrupt or lose it.

- **Loss of privacy**: the cloud provider has access to both the stored data and how it is accessed. The provider may be trustworthy, but malicious insiders are a wide-spread security problem [HDS+11]. This is an especial concern in applications that involve keeping private data like health records. An obvious solution is the customer encrypting the data before storing it, but if the data is accessed by distributed applications this involves running protocols for key distribution (processes in different machines need access to the cryptographic keys). DEPSKY employs a secret sharing scheme and erasure codes to avoid storing clear data in the clouds and to improve the storage efficiency, amortizing the replication factor on the cost of the solution.

- **Vendor lock-in**: there is currently some concern that a few cloud computing providers may become dominant, the so called vendor lock-in issue [ALPW10]. This concern is specially prevalent in Europe, as the most conspicuous providers are not in the region. Even moving from one provider to another one may be expensive because the cost of cloud usage has a component proportional to the amount of data that is read and written. DEPSKY addresses this issue in two ways. First, it does not depend on a single cloud provider, but on a few, so data access can be balanced among the providers considering their practices (e.g., what they charge). Second, DEPSKY uses erasure codes to store only a fraction (typically half) of the total amount of data in each cloud. In case the need of exchanging one provider by another arises, the cost of migrating the data will be at most a fraction of what it would be otherwise.

The way in which DEPSKY solves these limitations does not come for free. At first sight, using, say, four clouds instead of one involves costs roughly four times higher. One of the key objectives of DEPSKY is to reduce this cost, which in fact it does to about 1.2 to 2 times the cost of using a single cloud. This seems to be a reasonable cost, given the benefits.

The key insight of this work is that these limitations of individual clouds can be overcome by using a *cloud-of-clouds* in which the operations (read, write, etc.) are implemented using a set of *Byzantine quorum systems protocols*. The protocols require *diversity* of location, administration, design and implementation, which in this case comes directly from the use of different commercial clouds [Vuk10]. There are protocols of this kind in the literature, but they either require that the servers execute some protocol-specific code [CT06, GWGR04, MR97, MR98, MAD02], not possible in storage clouds, or are sensible to contention (e.g., [ACKM06]), which makes them difficult to use for geographically dispersed systems with high and variable access latencies. DEPSKY overcomes these limitations by not requiring specific code execution in the servers (i.e., storage clouds), but still being efficient by requiring only two communication round-trips for each operation. Furthermore, it leverages the above mentioned mechanisms to deal with data confidentiality and reduce the amount of data stored in each cloud.

Although DEPSKY is designed for data replication on cloud storage systems, the weak assumptions required by its protocols make it usable to replicate data on arbitrary storage systems such as FTP servers and key-value databases. This extended applicability is only possible because, as already mentioned, DEPSKY protocols have no server-side specific code to be executed, requiring only basic storage operations to write, read and list objects.

In summary, the main contributions of work presented in this chapter are:

1. The DEPSKY system, a storage cloud-of-clouds that overcomes the limitations of individual clouds by using an efficient set of Byzantine quorum system protocols, cryptography, secret sharing, erasure codes and the diversity that comes from using several clouds. The

DEPSKY protocols require at most two communication round-trips for each operation and store only approximately half of the data in each cloud for the typical case.

2. The notion of consistency proportional storage, in which the replicated storage system provides the same consistency semantics as its base objects (i.e., the nodes where the data is stored). DEPSKY satisfies this property for a large spectrum of consistency models, encompassing most of the semantics provided by storage clouds and popular storage systems.

3. A set of experiments showing the costs and benefits (both monetary and in terms of performance) of storing updatable data blocks in more than one cloud. The experiments were made during one month, using four commercial cloud storage services (Amazon S3, Windows Azure Blob Service, Nirvanix CDN and Rackspace Files) and PlanetLab to run clients that access the service from several places worldwide.

The chapter is organized as follows. Section 4.2 describe some applications that can make use of DEPSKY. Section 4.3 presents the core protocols employed in our system and Section 4.4 presents additional protocols for locking and management operations. Sections 4.5 and 4.6 show how storage clouds access control can be employed to setup a DEPSKY cloud-of-clouds storage and how the system works with weakly consistent clouds, respectively. The description of the DEPSKY implementation and its experimental evaluation are presented in Sections 4.7 and 4.8. Finally, Section 4.9 discusses related work and Section 4.10 presents some final remarks. Additional related material appear on Appendix A. This material describes some auxiliary functions used in our algorithms (Section A.1), the correctness proofs for the storage (Section A.2) and locking protocols (Section A.3) and a proof of the DEPSKY consistency proportionality (SectionA.4).

## 4.2 Cloud Storage Applications

Examples of applications that can benefit from DEPSKY are the following:

**Critical data storage.** Given the overall advantages of using clouds for running large scale systems, many governments around the globe are considering the use of this model. Recently, the US government announced its interest in moving some of its computational infrastructure to the cloud and started some efforts in understanding the risks involved in doing these changes [Gre10]. The European Commission is also investing in the area through FP7 projects like TClouds.

In the same line of these efforts, there are many critical applications managed by companies that have no interest in maintaining a computational infrastructure (i.e., a datacenter). For these companies, the cloud computing pay-per-use model is specially appealing. An example would be power system operators. Considering only the case of storage, power systems have data historian databases that store events collected from the power grid and other subsystems. In such a system, the data should be always available for queries (although the workload is mostly write-dominated) and access control is mandatory.

Another critical application that could benefit from moving to the cloud is a unified medical records database, also known as electronic health record (EHR). In such an application, several hospitals, clinics, laboratories and public offices share patient records in order to offer a better service without the complexities of transferring patient information between them. A system

like this has been being deployed in the UK for some years [Ehs10]. Similarly to our previous example, availability of data is a fundamental requirement of a cloud-based EHR system, and privacy concerns are even more important.

A somewhat related example comes from the observation that some biomedical companies that generate high-value data would not put it on a third party cloud without ensuring confidentiality. In fact, some of these companies are actively stripping biomedical data stored on several clouds to avoid complete confidentiality loss in case of cloud compromise [May10].

All these applications can benefit from a system like DEPSKY. First, the fact that the information is replicated on several clouds would improve the data availability and integrity. Moreover, the DEPSKY-CA protocol (Section 4.3) ensures the confidentiality of stored data and therefore addresses some of the privacy issues so important for these applications. Finally, these applications are prime examples of cases in which the extra costs due to replication are affordable for the added quality of service since the amount of data stored is not large when compared with Internet-scale services.

Notice that the application domains described above are represented in TClouds projects through WP3.1 and 3.2.

**Content distribution.** One of the most surprising uses of Amazon S3 is content distribution [Hen09]. In this scenario, users use the storage system as distribution points for their data in such a way that one or more producers store the content on their account and a set of consumers read this content. A system like DEPSKY that supports dependable updatable information storage can help this kind of application when the content being distributed is dynamic and there are security concerns associated. For example, a company can use the system to give detailed information about its business (price, available stock, etc.) to its affiliates with improved availability and security.

**Future applications.** Many applications are moving to the cloud, so, it is possible to think of new applications that would use the storage cloud as a back-end storage layer. Relational databases [BFG+08], file systems [VSV12], objects stores and key-value databases are example of systems that can use the cloud as storage layer as long as caching and weak consistency models [TDP+94, Vog09] are used to avoid paying the price of cloud access on every operation.

## 4.3 The DEPSKY System

This section describes the DEPSKY system. It presents the system architecture, the data and system models, the protocol design rational, the two main protocols (DEPSKY-A and DEPSKY-CA), and some optimizations.

### 4.3.1 DEPSKY Architecture

Figure 4.3.1 presents the architecture of DEPSKY. As mentioned before, the clouds are storage clouds *without the capacity of executing users' code*, so they are accessed using their standard interface without modifications. The DEPSKY algorithms are implemented as a software library in the clients. This library offers an *object store* interface [GNA+98], similar to what is used by parallel file systems (e.g., [GGL03, WBM+06]), allowing reads and writes in the back-end (in this case, the untrusted clouds).

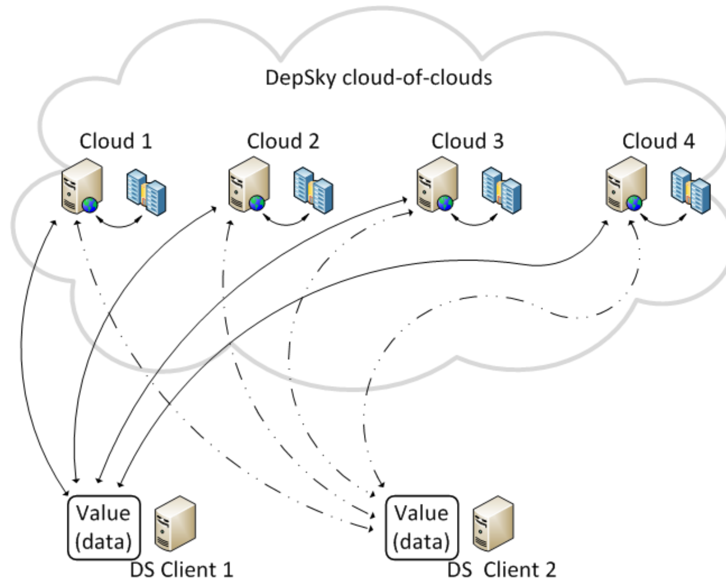



Figure 4.1: Architecture of DEPSKY (with 4 clouds and 2 clients).

### 4.3.2 Data Model

The use of diverse clouds requires the DEPSKY library to deal with the heterogeneity of the interfaces of each cloud provider. An aspect that is specially important is the format of the data accepted by each cloud. The data model allow us to ignore these details when presenting the algorithms.

Figure 4.3.2 presents the DEPSKY data model with its three abstraction levels. In the first (left), there is the *conceptual data unit*, which corresponds to the basic storage object with which the *algorithms* work (a register in distributed computing parlance [Lam86, MR97]). A data unit has a unique name (X in the figure), a version number (to support updates on the object), verification data (usually a cryptographic hash of the data) and the data stored on the data unit object. In the second level (middle), the conceptual data unit is implemented as a *generic data unit* in an *abstract storage cloud*. Each generic data unit, or *container*, contains two types of files: a signed metadata file and the files that store the data. Metadata files contain the version number and the verification data, together with other information that applications may demand. Notice that a data unit (conceptual or generic) can store several versions of the data, i.e., the container can contain several data files. The name of the metadata file is simply *metadata*, while the data files are called *value<Version>*, where <Version> is the version number of the data (e.g., *value1*, *value2*, etc.). Finally, in the third level (right) there is the *data unit implementation*, i.e., the container translated into the specific constructions supported by each cloud provider (Bucket, Folder, etc.). Notice that the one-container-per-data-unit policy may be difficult to implement in some clouds (e.g., Amazon S3 has a limit of 100 buckets per account, limiting the system to 100 data units). However, it is possible to store several data units on the same container as long as the data unit name is used as a prefix of their files names.

The data stored on a data unit can have arbitrary size, and this size can be different for different versions. Each data unit object supports the usual object store operations: creation (create the container and the metadata file with version 0), destruction (delete or remove access to the data unit), write and read.

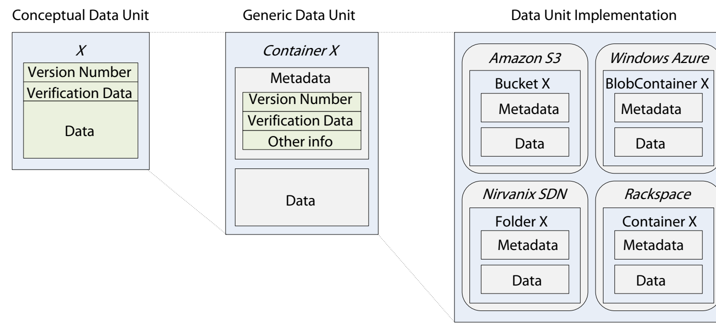



Figure 4.2: DEPSKY data unit and the 3 abstraction levels.

### 4.3.3 System Model

We consider an *asynchronous distributed system* composed by three types of parties: writers, readers and cloud storage providers. The latter are the clouds 1-4 in Figure 4.3.1, while writers and readers are roles of the clients, not necessarily different processes.

**Readers and writers.** Readers can fail arbitrarily, i.e., they can crash, fail intermittently and present any behavior. Writers, on the other hand, are assumed to fail only by crashing. We do not consider that writers can fail arbitrarily because, even if the protocol tolerated inconsistent writes in the replicas, faulty writers would still be able to write wrong values in data units, effectively corrupting the state of the application that uses DEPSKY. Moreover, the protocols that tolerate malicious writers are much more complex (e.g., [CT06,LR06]), with active servers verifying the consistency of writer messages, which cannot be implemented on general storage clouds (Section 4.3.4).

All writers of a data unit $du$ share a common private key $K_{r_w}^{du}$ used to sign some of the data written on the data unit (function $sign(DATA, K_{r_w}^{du})$), while readers of $du$ have access to the corresponding public key $K_{u_w}^{du}$ to verify these signatures (function $verify(DATA, K_{u_w}^{du})$). This public key can be made available to the readers through the storage clouds themselves. Moreover, we assume also the existence of a collision-resistant *cryptographic hash function* H.

**Cloud storage providers.** Each cloud is modeled as a *passive storage entity* that supports five operations: $list$ (lists the files of a container in the cloud), $get$ (reads a file), $create$ (creates a container), $put$ (writes or modifies a file in a container) and $remove$ (deletes a file). By passive storage entity, we mean that no protocol code other than what is needed to support the aforementioned operations is executed. We assume that access control is provided by the clouds in order to ensure that readers are only allowed to invoke the list and get operations (more about it in Section 4.5).

Since we do not trust clouds individually, we assume they can fail in a Byzantine way [LSP82]: data stored can be deleted, corrupted, created or leaked to unauthorized parties. This is the most general fault model and encompasses both malicious attacks/intrusions on a cloud provider and arbitrary data corruption (e.g., due to accidental events like the Ma.gnolia case). The protocols require a set of $n = 3f + 1$ storage clouds, at most $f$ of which can be faulty. Additionally, the quorums used in the protocols are composed by any subset of $n - f$ storage clouds. It is worth to notice that this is the minimum number of replicas to tolerate Byzantine servers in asynchronous storage systems [MAD02].

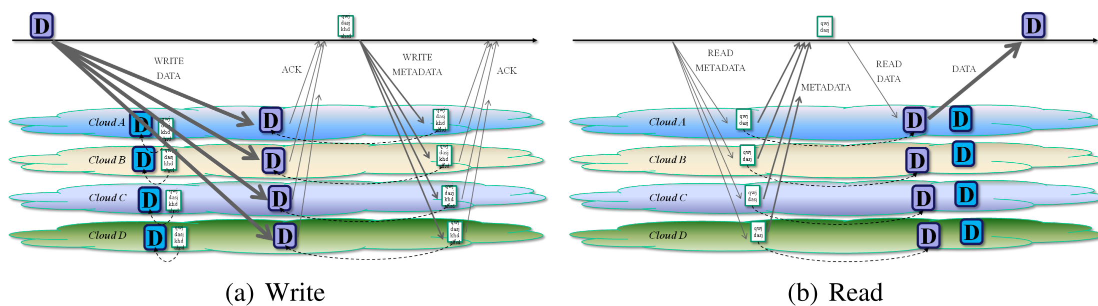
Readers, writers and clouds are said to be *correct* if they do not fail.

The register abstraction provided by DEPSKY satisfies a semantics that depends on the semantics provided by the underlying clouds. For instance, if the $n$ clouds provide regular semantics, then DEPSKY also satisfies regular semantics: a read operation that happens concurrently with a write can return the value being written or the object's value before the write [Lam86]. We discuss the semantics of DEPSKY in detail in Section 4.6.

Notice that our model hides most of the complexity of the distributed storage system employed by the cloud provider: it just assumes that this system is an object storage prone to Byzantine failures that supports very simple operations. These operations are accessed through RPCs (Remote Procedure Calls) with the following failure semantics: the operation keeps being invoked until a reply is received or the operation is canceled (possibly by another thread, using a $cancel_pending$ special operation to stop resending a request). This means that we have at most once semantics for the operations being invoked. Repeating the operation is not a problem because all storage cloud operations are idempotent, i.e., the state of the cloud becomes the same irrespectively of the operation being executed only once or more times.

### 4.3.4 Protocol Design Rationale

Quorum protocols can serve as the backbone of highly available storage systems [CGKV09]. There are many quorum protocols for implementing Byzantine fault-tolerant (BFT) storage [CT06, GWGR04, HGR07, LR06, MR97, MR98, MAD02], but most of them require that the servers execute protocol-specific code, a functionality not available on storage clouds. In consequence, cloud-specific protocols need to assume *passive storage replicas*, supporting only (blind) reads and writes. This leads to a key difference between the DEPSKY protocols and these classical BFT quorum protocols: *metadata and data are written and read in separate quorum accesses*. Moreover, these two accesses occur in different orders on read and write protocols, as depicted in Figure 4.3.4. This feature is crucial for the protocol correctness and efficiency.




(a) Write (b) Read

Figure 4.3: DEPSKY read and write protocols.

Supporting multiple writers for a register (a data unit in DEPSKY parlance) can be problematic due to the lack of server code able to verify the version number of the data being written. To overcome this limitation we implement a single-writer multi-reader register, which is sufficient for many applications, and we provide a lock/lease protocol to support several concurrent writers for the data unit. However, the next chapter (Chapter 5) study the costs of supporting multi-writer storage in the cloud-of-clouds model (with non-Byzantine storage providers). As can be seen there, this feature is possible, but it incurs some added performance penalties that are, in principle, unavoidable.

There are also some quorum protocols that consider individual storage nodes as passive shared memory objects (or disks) instead of servers [ACKM06, AL03, CM02, GL03, JCT98]. Unfortunately, most of these protocols require many steps to access the shared memory, or are heavily influenced by contention, which makes them impractical for geographically dispersed distributed systems such as DEPSKY due to the highly variable latencies involved. As show in Figure 4.3.4, DEPSKY protocols require two communication round-trips to read or write the metadata and the data files that are part of the data unit, independently of the existence of faults and contention.

Furthermore, as will be discussed latter, many clouds do not provide the expected consistency guarantees of a disk, something that can affect the correctness of these protocols. The DEPSKY protocols provide *consistency-proportional semantics*, i.e., the semantics of a data unit is as strong as the underling clouds allow, from eventual to regular consistency semantics. We do not try to provide atomic (linearizable) semantics [Lam86, HW90] due to the fact that all known techniques require server-to-server communication [CT06], servers sending update notifications to clients [MAD02] or write-backs [GWGR04, MR98]. None of these mechanisms is implementable using general-purpose storage clouds.

To ensure the confidentiality of the data stored in the clouds we encrypt it using symmetric cryptography. To avoid the need of a key distribution service, which would have to be implement outside of the clouds, we employ a *secret sharing scheme* [Sha79]. In this scheme, a dealer (the writer in the case of DEPSKY) distributes a secret (the encryption key) to $n$ players (clouds in our case), but each player gets only a share of this secret. The main properties of the scheme is that at least $f + 1 \leq n - f$ different shares of the secret are needed to recover it and that no information about the secret is disclosed with $f$ or less shares. The scheme is integrated on the basic replication protocol in such way that each cloud stores just a share of the key used to encrypt the data being written. This ensures that no individual cloud will have access to the encryption key. On the contrary, clients that have authorization to access the data will be granted access to the key shares of (at least) $f + 1$ different clouds, so they will be able to rebuild the encryption key and decrypt the data.

The use of a secret sharing scheme allows us to integrate confidentiality guarantees to the stored data without using a key distribution mechanism to make writers and readers of a data unit share a secret key. In fact, our mechanism reuses the access control of the cloud provider to control which readers are able to access the data stored on a data unit.

Although it may seem questionable if avoiding key distribution methods is useful for a large spectrum of applications, our previous experience with secret sharing schemes [BACF08] suggests that the overhead of using them is not deterrent, specially if one considers the communication latency of accessing a cloud storage provider. However, the protocol can be easily modified to use a shared key for confidentiality if such key distribution method is available.

If we simply replicate the data on $n$ clouds, the monetary costs of storing data using DEPSKY would increase by a factor of $n$. In order to avoid this, we compose the secret sharing scheme used on the protocol with an *information-optimal erasure code algorithm*, reducing the size of each share by a factor of $\frac{n}{f+1}$ of the original data [Rab89]. This composition follows the original proposal of [Kra93], where the data is encrypted with a random secret key, the encrypted data is encoded, the key is divided using secret sharing and each server receives a block of the encrypted data and a share of the key.

Common sense says that for critical data it is always a good practice not erasing all old versions of the data, unless we can be certain that we will not need them anymore [Ham07]. An additional feature of our protocols is that old versions of the data are kept in the clouds unless they are explicitly deleted.

### 4.3.5 DEPSKY-A– Available DepSky

The first DEPSKY protocol is called DEPSKY-A. It improves the availability and integrity of cloud-stored data by replicating it on several clouds using quorum techniques. Algorithm 1 presents this protocol. We encapsulate some of the protocol steps in the functions described in Table 4.1. We use the '.' operator to denote access to metadata fields, e.g., given a metadata file $m$, $m.ver$ and $m.digest$ denote the version number and digest(s) stored in $m$. We use the '+' operator to concatenate two items into a string, e.g., "value-"+*new_ver* produces a string that starts with the string "value-" and ends with the value of variable *new_ver* in string format. Finally, the $max$ function returns the maximum among a set of numbers.

Table 4.1: Functions used in the DEPSKY-A protocols (implementation in Appendix A.1).

| Function | Description |
|---|---|
| $queryMetadata(du)$ | obtains the correctly signed file metadata stored in the container $du$ of $n-f$ clouds used to store the data unit and returns it in an array |
| $writeQuorum(du, name, value)$ | for every cloud $i \in \{0, ..., n-1\}$, writes the $value[i]$ on a file named $name$ on the container $du$ in that cloud and waits for write confirmations from $n-f$ clouds |

The key idea of the *write algorithm* (lines 1-13) is to first write the value in a quorum of clouds (line 8), then write the corresponding metadata (line 12), as ilustrated in Figure 4.3(a). This order of operations ensures that a reader will only be able to read metadata for a value already stored in the clouds. Additionally, when a writer first writes a data unit $du$ (lines 3-5, $max_ver_{du}$ initialized with 0), it first contacts the clouds to obtain the metadata with the greatest version number, then updates the $max_ver_{du}$ variable with the current version of the data unit.

The *read algorithm* starts by fetching the metadata files from a quorum of clouds (line 16) and choosing the one with greatest version number (line 17). After that, the algorithm enters in a loop where it keeps looking at the clouds until it finds the data unit version corresponding to this version number and the cryptographic hash found in the chosen metadata (lines 18-26). Inside of this loop, the process fetches the file from the clouds until either it finds one value file containing the value matching the digest on the metadata or the value is not found on at least $n-f$ clouds[1] (lines 20-24). Finally, when a valid value is read, the reader cancels the pending RPCs, exits the loop and returns the value (lines 25-27). The normal case execution (with some optimizations discussed in Section 4.3.7) is illustrated in Figure 4.3(b).

The rationale of why this protocol provides the desired properties is the following (proofs in the Appendix A.2). Availability is guaranteed because the data is stored in a quorum of at least $n-f$ clouds and it is assumed that at most $f$ clouds can be faulty. The read operation has to retrieve the value from only one of the clouds (line 22), which is always available because $(n-f)-f > 1$ . Together with the data, signed metadata containing its cryptographic hash is also stored. Therefore, if a cloud is faulty and corrupts the data, this is detected when the metadata is retrieved. Moreover, the fact that metadata files are self-verifiable (i.e., signed) and quorums overlap in at least $f+1$ clouds (one correct) ensures the last written metadata file will be read. Finally, the outer loop of the read ensures that the read of a value described on a read

[1]This is required to avoid the process to block forever waiting replies from $f$ faulty clouds.

**Algorithm 1:** DEPSKY-A read and write protocols.

```
1  procedure DepSkyAWrite(du,value)
2  begin
3      if max_ver_du = 0 then
4          m ⟵ queryMetadata(du)
5          max_ver_du ⟵ max({m[i].ver : 0 ≤ i ≤ n − 1})
6      new_ver ⟵ max_ver_du + 1
7      v[0 .. n − 1] ⟵ value
8      writeQuorum(du,"value-"+new_ver, v)
9      new_meta ⟵ ⟨new_ver, H(value)⟩
10     sign(new_meta, K_rw^du)
11     v[0 .. n − 1] ⟵ new_meta
12     writeQuorum(du,"metadata", v)
13     max_ver_du ⟵ new_ver

14 function DepSkyARead(du)
15 begin
16     m ⟵ queryMetadata(du)
17     max_id ⟵ i : m[i].ver = max({m[i].ver : 0 ≤ i ≤ n − 1})
18     repeat
19         v[0 .. n − 1] ⟵ ⊥
20         parallel for 0 ≤ i < n − 1 do
21             tmp_i ⟵ cloud_i.get(du,"value-" +m[max_id].ver)
22             if H(tmp_i) = m[max_id].digest then v[i] ⟵ tmp_i
23             else v[i] ⟵ ERROR
24         wait until (∃i : v[i] ≠⊥ ∧v[i] ≠ ERROR) ∨ (|{i : v[i] ≠⊥}| ≥ n − f)
25         for 0 ≤ i ≤ n − 1 do cloud_i.cancel_pending()
26     until ∃i : v[i] ≠⊥ ∧v[i] ≠ ERROR
27     return v[i]
```

metadata will be repeated until it is available, which will eventually holds since a metadata file is written only after the data file is written.

### 4.3.6 DEPSKY-CA– Confidential & Available DepSky

The DEPSKY-A protocol has two main limitations. First, a data unit of size S consumes $n \times S$ storage capacity of the system and costs on average $n$ times more than if it was stored in a single cloud. Second, it stores the data in cleartext, so it does not give confidentiality guarantees. To cope with these limitations we employ an information-efficient secret sharing scheme [Kra93] that combines symmetric encryption with a classical secret sharing scheme and an optimal erasure code to partition the data in a set of blocks in such a way that (i.) $f + 1$ blocks are necessary to recover the original data and (ii.) $f$ or less blocks do not give any information about the stored data[2]. The overall process is illustrated in Figure 4.3.6.

The DEPSKY-CA protocol integrates these techniques with the DEPSKY-A protocol (Algorithm 2). The additional cryptographic and coding functions needed are in Table 4.2. The differences of DEPSKY-CA protocol in relation to DEPSKY-A are the following: (1.) the en-

[2]Erasure codes alone cannot satisfy this confidentiality guarantee.

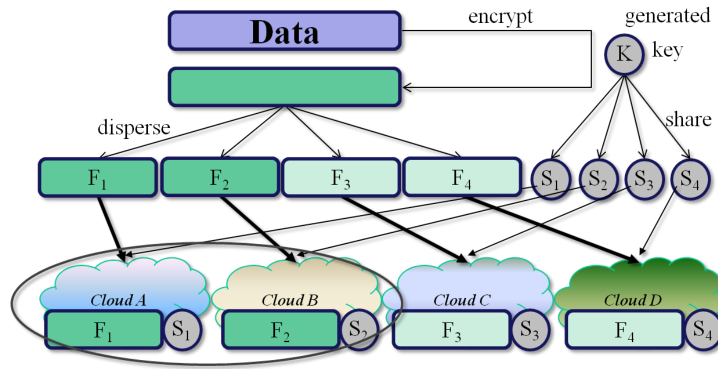



Figure 4.4: The combination of symmetric encryption, secret sharing and erasure codes in DEPSKY-CA.

cryption of the data, the generation of the key shares and the encoding of the encrypted data on DepSkyCAWrite (lines 7-10) and the reverse process on DepSkyCARead (lines 33-35), as show in Figure 4.3.6; (2.) the data stored in $cloud_i$ is composed by the share of the key $s[i]$ and the encoded block $v[i]$ (line 12); and (3.) $f + 1$ replies are necessary to read the data unit's current value instead of one on DEPSKY-A (lines 30 and 32). Additionally, instead of storing a single digest on the metadata file, the writer generates and stores $n$ digests, one for each cloud. These digests are accessed as different positions of the $digest$ field of a metadata. If a key distribution infrastructure is available, or if readers and writer share a common key $k$, the secret sharing scheme can be removed (lines 7, 9 and 34 are not necessary).

The rationale of the correctness of the protocol is similar to the one for DEPSKY-A (proofs also in the Appendix A.2). The main differences are those already pointed out: encryption prevents individual clouds from disclosing the data; secret sharing allows storing the encryption key in the cloud without $f$ faulty clouds being able to reconstruct it; the erasure code scheme reduces the size of the data stored in each cloud.

### 4.3.7 Optimizations

This section introduces two optimizations that can make the protocols more efficient and cost-effective. In Section 4.8 we evaluate the impact of these optimizations on the protocols.

**Write.** In the DEPSKY-A and DEPSKY-CA write algorithms, a value file is written using the function $writeQuorum$ (see Table 4.1). This function tries to write the file on all clouds and waits for confirmation from a quorum. A more cost-effective solution would be to try to store the value only on a *preferred quorum*, resorting on extra clouds only if the reception of write confirmations from the quorum of clouds is not received until a timeout. This optimization can be applied both to DEPSKY-A and DEPSKY-CA to make the data be stored only in $n - f$ out-of $n$ clouds, which can decrease the DEPSKY storage cost by a factor of $\frac{n-f}{n}$, possibly with some loss in terms of availability and durability of the data.

**Read.** The DEPSKY-A algorithm described in Section 4.3.5 tries to read the most recent version of the data unit from all clouds and waits for the first valid reply to return it. In the pay-per-use model this is far from ideal because the user will pay for $n$ data accesses. A lower-cost solution is to use some criteria to sort the clouds and try to access them sequentially, one

**Algorithm 2:** DEPSKY-CA read and write protocols.

1 **procedure** DepSkyCAWrite(du,value)
2 **begin**
3  **if** $max_ver_{du} = 0$ **then**
4   $m \longleftarrow queryMetadata(du)$
5   $max_ver_{du} \longleftarrow \max(\{m[i].version\ :\ 0 \leq i \leq n-1\})$
6  $new_ver \longleftarrow max_ver_{du} + 1$
7  $k \longleftarrow generateSecretKey()$
8  $e \longleftarrow \mathsf{E}(value, k)$
9  $s[0 \mathinner{..} n-1] \longleftarrow share(k, n, f+1)$
10  $v[0 \mathinner{..} n-1] \longleftarrow encode(e, n, f+1)$
11  **for** $0\ \leq\ i\ <\ n-1$ **do**
12   $d[i] \longleftarrow \langle s[i], v[i] \rangle$
13   $h[i] \longleftarrow \mathsf{H}(d[i])$
14  $writeQuorum(du,$“value-”$+new_ver, d)$
15  $new_meta \longleftarrow \langle new_ver, h \rangle$
16  $sign(new_meta, K^{du}_{rw})$
17  $v[0 \mathinner{..} n-1] \longleftarrow new_meta$
18  $writeQuorum(du,$“metadata”$, v)$
19  $max_ver_{du} \longleftarrow new_ver$

20 **function** DepSkyCARead(du)
21 **begin**
22  $m \longleftarrow queryMetadata(du)$
23  $max_id \longleftarrow i\ :\ m[i].ver\ = \max(\{m[i].ver\ :\ 0 \leq i \leq n-1\})$
24  **repeat**
25   $d[0 \mathinner{..} n-1] \longleftarrow \perp$
26   **parallel for** $0\ \leq\ i\ \leq\ n-1$ **do**
27    $tmp_i \longleftarrow cloud_i.get(du,$ “value-” $+m[max_id].ver)$
28    **if** $\mathsf{H}(tmp_i) = m[max_id].digest[i]$ **then** $d[i] \longleftarrow tmp_i$
29    **else** $d[i] \longleftarrow ERROR$
30   **wait until** $(|\{i\ :\ d[i] \neq \perp \wedge d[i] \neq ERROR\}| > f) \vee (|\{i\ :\ d[i] \neq \perp\}| > n-f)$
31   **for** $0\ \leq\ i\ \leq\ n-1$ **do** $cloud_i.cancel_pending()$
32  **until** $|\{i\ :\ d[i] \neq \perp \wedge d[i] \neq ERROR\}| > f$
33  $e \longleftarrow decode(d.e, n, f+1)$
34  $k \longleftarrow combine(d.s, n, f+1)$
35  **return** $\mathsf{D}(e, k)$

at time, until the value is obtained. The sorting criteria can be based on access monetary cost (cost-optimal), the latency of $queryMetadata$ on the protocol (latency-optimal), a mix of the two or any other more complex criteria (e.g., an history of the latency and faults of the clouds).

This optimization can also be used to decrease the monetary cost of the DEPSKY-CA read operation. The main difference is that instead of choosing one of the clouds at a time to read the data, $f+1$ of them are chosen.

Table 4.2: Functions used in the DEPSKY-CA protocols.

| Function | Description |
| --- | --- |
| $generateSecretKey()$ | generates a random secret key |
| $\mathsf{E}(v,k)$/$\mathsf{D}(e,k)$ | encrypts $v$ and decrypts $e$ with key $k$ |
| $encode(d,n,t)$ | encodes $d$ on $n$ blocks in such a way that $t$ are required to recover it |
| $decode(db,n,t)$ | decodes array $db$ of $n$ blocks, with at least $t$ valid, to recover $d$ |
| $share(s,n,t)$ | generates $n$ shares of $s$ in such a way that at least $t$ of them are required to obtain any information about $s$ |
| $combine(ss,n,t)$ | combines shares on array $ss$ of size $n$ containing at least $t$ correct shares to obtain the secret $s$ |

## 4.4 DEPSKY Extensions

In this section we present a set of additional protocols that may be useful for implementing real systems using DEPSKY.

### 4.4.1 Supporting Multiple Writers – Locking with Storage Clouds

The DEPSKY protocols presented do not support concurrent writes, which is sufficient for many applications where each process writes on its own data units. However, there are applications in which this is not the case. An example is a fault-tolerant storage system that uses DEPSKY as its back-end object store. This system could have more than one node with the writer role writing in the same data unit(s) for fault tolerance reasons. If the writers are in the same network, coordination services like ZooKeeper [HKJR10] or DepSpace [BACF08] can be used to elect a leader and coordinate the writes. However, if the writers are scattered through the Internet this solution is not practical without trusting the site in which the coordination service is deployed (and even in this case, the coordination service may be unavailable due to network issues). Open coordination services such as WSDS [ABF08] can still be used, but they require an Internet deployment.

The solution we advocate is a *low contention lock mechanism* that uses the cloud-of-clouds itself to maintain lock files on a data unit. These files specify which is the writer and for how much time it has write access to the data unit. However, for this solution to work, two additional assumptions must hold. The first one is related with the use of leases. The algorithm requires every contending writer to have *synchronized clocks* with a precision of $\Delta$. This can be ensured in practice by making all writers that want to lock a data unit synchronize their clocks with a common NTP (Network Time Protocol [Mil92]) server with a precision of $\frac{\Delta}{2}$. The second assumption is related with the consistency of the clouds. We assume *regular semantics* [Lam86] for the creation and listing of files on a container (which are equivalent to write and read operations, respectively). Although this assumption appears to be too strong, object storage services like Amazon S3 already ensure this kind of consistency for object creation, sometimes called *read-after-write* [f W11]. Anyway, in Section 4.6 we discuss the effects of weakly consistent clouds on this protocol.

The lock protocol is described in Algorithm 3, and it works as follows. A process $c$ that wants to be a writer (and has permission to be), first lists files on the data unit container on a quorum of clouds and tries to find a valid file called *lock-c'-T'* with $c' \neq c$ and local time on the process smaller than $T' + \Delta$ (lines 5-10). If such file is found in some cloud, it means that some other process $c'$ holds the lock for this data unit and $c$ will sleep for a random amount of time before trying to acquire the lock again (line 21). If the file is not found, $c$ can write a lock file named *lock-c-T* containing a cryptographic signature of the file name on all clouds (lines 11 and 12), being $T = local_clock + LEASE_TIME$. In the last step, $c$ lists again all files in the data unit container searching for valid and not expired lock files from other processes (lines 13-17). If a file like that is found, $c$ removes the lock file it wrote from the clouds and sleeps for a small random amount of time before trying to run the protocol again (lines 18-21). Otherwise, $c$ becomes the single-writer for the data unit until $T$.

The protocol also uses a predicate $valid$ that verifies if the lock file was not created by a faulty cloud. The predicate is true if the lock file is returned by either $f + 1$ clouds or its contents is correctly signed by its owner (line 28).

Several remarks can be made about this protocol. First, the backoff strategy is necessary to ensure that two processes trying to become writers at the same time never succeed. Second, locks can be renewed periodically to ensure existence of a single writer at every moment of the execution. Unlocking can be easily done through the removal of the lock files (lines 24-27). Third, this lock protocol is only *obstruction-free* [HLM03]: if several process try to become writers at the same time, it is possible that none of them are successful. However, due to the backoff strategy used, this situation should be very rare on the envisioned deployments. Finally, it is important to notice that the unlock procedure is not fault-tolerant: in order to release a lock, the lock file has to be deleted from all clouds; a malicious cloud can still show the removed lock file disallowing lock acquisition by other writers. However, given the finite validity of a lock, this problem can only affect the system for a limited period of time, after which the problematic lock expires.

The proof that this protocol satisfies mutual exclusion and obstruction-freedom is presented in Appendix A.3.

### 4.4.2 Management Operations

Besides read, write and lock, DEPSKY provides other operations to manage data units. These operations and underlying protocols are briefly described in this section.

**Creation and destruction.** The creation of a data unit can be easily done through the invocation of the create operation in each individual cloud. In contention-prone applications, the creator should execute the locking protocol of the previous section before executing the first write to ensure it is the single writer of the data unit.

The destruction of a data unit is done in a similar way: the writer simply removes all files and the container that stores the data unit by calling $remove$ in each individual cloud.

**Garbage collection.** As already discussed in Section 4.3.4, we choose to keep old versions of the value of the data unit on the clouds to improve the dependability of the storage system. However, after many writes the amount of storage used by a data unit can become very high and thus some garbage collection is necessary. The protocol for doing that is very simple: a writer just lists all files named “value-*version*” in the data unit container and removes all those with *version* smaller than the oldest version it wants to keep in the system.

**Algorithm 3:** DEPSKY data unit locking by writer $c$.

```
1  function DepSkyLock(du)
2  begin
3      lock_id ←⊥
4      repeat
           // list lock files on all clouds to see if the du is locked
5          L[0 .. n − 1] ←−⊥
6          parallel for 0 ≤ i ≤ n − 1 do
7              L[i] ←− cloud_i.list(du)
8          wait until (|{i : L[i] ≠⊥}| > n − f)
9          for 0 ≤ i ≤ n − 1 do cloud_i.cancel_pending()
10         if ∄i : ∃lock-c'-T'∈ L[i] : c' ≠ c ∧ valid(L,lock-c'-T', du) ∧ (T' + Δ > local_clock)
           then
               // create a lock file for the du and write it in the clouds
11             lock_id ←"lock-"+c+"-"+(local_clock + LEASE_TIME)
12             writeQuorum(du, lock_id, sign(lock_id, K^du_rc))
               // list the lock files again to detect contention
13             L[0 .. n − 1] ←−⊥
14             parallel for 0 ≤ i ≤ n − 1 do
15                 L[i] ←− cloud_i.list(du)
16             wait until (|{i : L[i] ≠⊥}| > n − f)
17             parallel for 0 ≤ i ≤ n − 1 do cloud_i.cancel_pending()
18             if ∃i : ∃lock-c'-T'∈ L[i] : c' ≠ c ∧ valid(L,lock-c'-T', du) ∧ (T' + Δ > local_clock)
               then
19                 DepSkyUnlock(lock_id)
20                 lock_id ←⊥
21         if lock_id =⊥ then sleep for some time
22     until lock_id ≠⊥
23     return lock_id

24 procedure DepSkyUnlock(lock_id)
25 begin
26     parallel for 0 ≤ i < n − 1 do
27         cloud_i.delete(du,lock_id)

28 predicate valid(L,lock-c'-T', du) ≡ (|{i : lock-c'-T'∈ L[i]}| > f ∨ verify(lock-c'-T', K^du_uc))
```

**Cloud reconfiguration.** Sometimes one cloud can become too expensive or too unreliable to be used for storing DEPSKY data units. For such cases DEPSKY provides a reconfiguration protocol that moves blocks from one cloud to another. The protocol is the following: (1.) the writer reads the data (probably from the other clouds and not from the one being removed); (2.) creates the data unit container on the new cloud; (3.) executes the write protocol on the clouds not removed and the new cloud; (4.) deletes the data unit from the cloud being removed. After that, the writer needs to inform the readers that the data unit location was changed. This can be done writing a special file on the data unit container of the remaining clouds informing the new configuration of the system. A process will read this file and accept the reconfiguration if this file is read from at least $f + 1$ clouds.

## 4.5 Cloud-of-Clouds Access Control

In this section we briefly discuss how cloud storage access control can be used to set up the access control for management, writers and readers of DEPSKY data units.

**Management.** All management operations described in Section 4.4.2 can only be executed by writers of a data unit, with the exception of the creation and destruction of a data unit, that needs to be carried on by the data unit' owner, that has write rights on the data unit container parent directory.

**Writers.** If a data unit has more than one possible writer, all of them should have the write rights on the data unit container. Moreover, all writers first write their public keys on the DU container before trying to acquire the lock for writing on the data unit. Notice that it is possible to have a single writer account, with a single shared writer private and public key pair, being used by several writer processes for fault tolerance reasons. Finally, when a writer does not need to write in a data unit anymore, it removes its public key from the data unit container on all clouds.

**Readers.** The readers of a data unit are defined by the set of accounts that have read access to the data unit container. It is worth to mention that some clouds such as Rackspace Files and Nirvanix CDN do not provide this kind of rich access control. These clouds only allow a file to be confidential (accessed only by its writer) or public (accessed by everyone that knows its URL). However, other popular storage clouds like Amazon S3, Windows Azure Blob Service and Google Docs support ACLs for giving read (and write) access to the files stored in a single account. We expect this kind of functionality to be available in most storage clouds in the near future.

Finally, all readers of a data unit consider that a metadata or lock file is correctly signed if the signature was produced with any of the writer keys available on the data unit container of $f + 1$ clouds.

## 4.6 Consistency Proportionality

Both DEPSKY-A and DEPSKY-CA protocols implement *single-writer multi-reader regular registers* if the clouds being accessed provide *regular semantics* [Lam86]. However, several clouds do not guarantee this semantics, but instead provide *read-after-write* (which is similar to the *safe semantics* [Lam86]) or *eventual consistency* [Vog09] for the data stored (e.g., Amazon S3 [Ama10]).

In fact, the DEPSKY read and write protocols are *consistency-proportional* in the following sense: *if the underlying clouds support a consistency model $\mathcal{C}$, the* DEPSKY *protocols provide consistency model $\mathcal{C}$*. This holds for any $\mathcal{C}$ among the following: *eventual* [Vog09], *read-your-writes*, *monotonic reads*, *writes-follow-reads*, *monotonic writes* [TDP+94] and *read-after-write* [Lam86]. A proof that DEPSKY provides consistency proportionality can be found in Appendix A.4.

Notice that if the underlying clouds are heterogeneous in terms of consistency guarantees, DEPSKY provides the weakest consistency among those provided. This comes from the fact that the consistency of a read directly depends of the reading of the last written metadata file.

Since we use read and write quorums with at least $f + 1$ clouds in their intersections, and since at most $f$ clouds may be faulty, the read of the most recently written metadata file may happen in the single correct cloud in such intersection. If this cloud does not provide strong consistency, the whole operation will be weakly consistent, following the consistency model of this cloud.

A problem with not having regular consistent clouds is that the lock protocol may not work correctly. After listing the contents of a container and not seeing a file, a process cannot conclude that it is the only writer. This problem can be minimized if the process waits a while between steps 2 and 3 of the protocol. However, the mutual exclusion guarantee will only be satisfied if the wait time is greater than the time for a data written to be seen by every other reader. Unfortunately, no eventually consistent cloud of our knowledge provides this kind of timeliness guarantee, but we can experimentally discover the amount of time needed for a read to propagate on a cloud with the desired coverage and use this value in the aforementioned wait. Moreover, to ensure some safety even when two writes happen in parallel, we can include a unique id of the writer (e.g., the hash of part of its private key) as the decimal part of its timestamps, just like it is done in most Byzantine quorum protocols (e.g., [MR97]). This simple measure allows the durability of data written by concurrent writers (the name of the data files will be different), even if the metadata file may point to different versions on different clouds.

## 4.7 DEPSKY Implementation

We have implemented a DEPSKY prototype in Java as an application library that supports the read and write operations. The code is divided in three main parts: (1) data unit manager, that stores the definition and information of the data units that can be accessed; (2) system core, that implements the DEPSKY-A and DEPSKY-CA read and write protocols; and (3) cloud drivers, which implements the logic for accessing the different clouds. The current implementation has 5 drivers available (the four clouds used in the evaluation - see next section - and one for storing data locally), but new drivers can be easily added. The overall implementation is about 2900 lines of code, being 1100 lines for the drivers.

The DEPSKY code follows a model of one thread per cloud per data unit in such a way that the cloud accesses can be executed in parallel (as described in the algorithms). All communications between clients and cloud providers are made over HTTPS (secure and private channels) using the REST APIs supplied by the storage cloud providers. Some of the clouds are accessed using the libraries available from the providers, called *access drivers*. To avoid problems due to the differences in implementation, in particular with different retransmission timeouts and retry policies, we disabled this feature from the drivers and implemented it on our code. The result is that all clouds are accessed using the same timeout and number of retries in case of failure.

The prototype employs *speculation* to execute the two phases of the read protocols in parallel. More precisely, as soon as a metadata file is read from a cloud $i$, the system starts fetching the data file from $i$, without waiting for $n - f$ metadata to find the one with greatest version number. The idea is to minimize access latency (which varies significantly in the different clouds) under the assumption that contention between reads and writes is rare and Byzantine faults seldom happen.

Our implementation makes use of several building blocks: RSA with 1024 bit keys for signatures, SHA-1 for cryptographic hashes, AES for symmetric cryptography, Shoenmakers' PVSS scheme [Sch99] for secret sharing with 192 bits secrets and the classic Reed-Solomon for erasure codes [Pla07]. Most of the implementations used come from the Java 6 API, while Java Secret Sharing [BACF08] and Jerasure [Pla07] were used for secret sharing and erasure

codes, respectively.

## 4.8 Evaluation

In this section we present an evaluation of DEPSKY which tries to answer three main questions: *What is the additional cost in using replication on storage clouds? What is the advantage in terms of performance and availability of using replicated clouds to store data? What are the relative costs and benefits of the two* DEPSKY *protocols?*

The evaluation focus on the case of $n = 4$ and $f = 1$, which we expect to be the common deployment setup of our system for two reasons: (1.) $f$ is the maximum number of faulty cloud storage providers, which are very resilient and so faults should be rare; (2.) there are currently not many more than four cloud storage providers that are adequate for storing critical data. Our evaluation uses the following cloud storage providers with their default configurations: Amazon S3, Windows Azure, Nirvanix and Rackspace.

### 4.8.1 Monetary cost evaluation

Storage cloud providers usually charge their users based on the amount of data uploaded, downloaded and stored on them. Table 4.3 presents the cost in US Dollars of executing 10,000 reads and writes using the DEPSKY data model (with metadata and supporting many versions of a data unit) considering three data unit sizes: 100kb, 1Mb and 10Mb. This table includes only the costs of the operations being executed (invocations, upload and download), not the data storage, which will be discussed latter. All estimations presented in this section were calculated based on the values charged by the four clouds at September 25th, 2010.

In the table, the columns "DS-A", "DS-A opt", "DS-CA" e "DS-CA opt" present the costs of using the DEPSKY protocols with the optimizations discussed in Section 4.3.7 disabled and enabled, respectively. The other columns present the costs for storing the data unit (DU) in a single cloud.

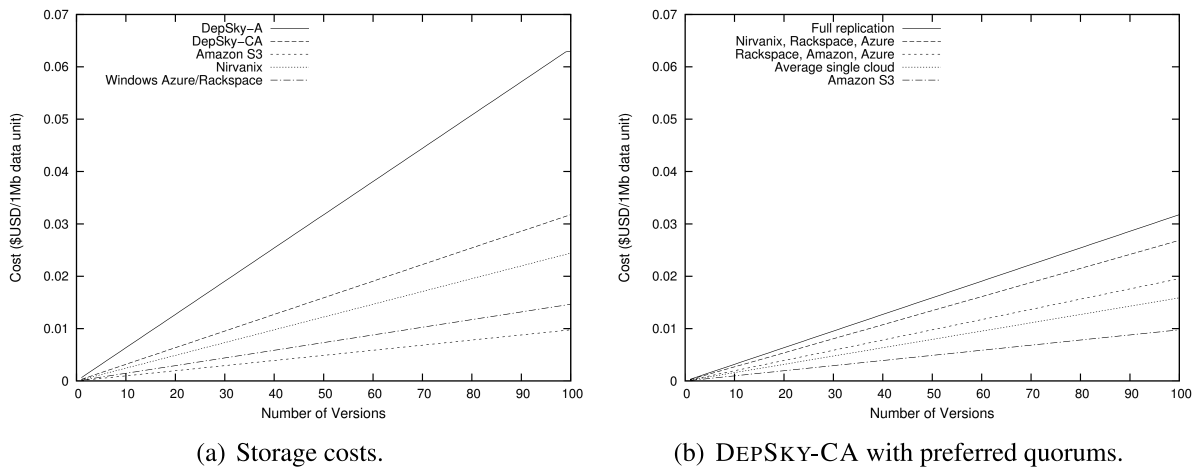
Table 4.3: Estimated costs per 10000 operations (in US Dollars). DEPSKY-A (DS-A)and DEPSKY-CA (DS-CA) costs are computed for the realistic case of 4 clouds ($f = 1$). The "DS-A opt" and "DS-CA opt" setups consider the cost-optimal version of the protocols with no failures.

| Operation | DU | DS-A | DS-A opt | DS-CA | DS-CA opt | Amazon | Rackspace | Azure | Nirvanix |
|---|---|---|---|---|---|---|---|---|---|
| 10K Reads | 100kb | 0.64 | 0.14 | 0.32 | 0.14 | 0.14 | 0.21 | 0.14 | 0.14 |
| | 1Mb | 6.55 | 1.47 | 3.26 | 1.47 | 1.46 | 2.15 | 1.46 | 1.46 |
| | 10Mb | 65.5 | 14.6 | 32.0 | 14.6 | 14.6 | 21.5 | 14.6 | 14.6 |
| 10K Writes | 100kb | 0.60 | 0.32 | 0.30 | 0.17 | 0.14 | 0.08 | 0.09 | 0.29 |
| | 1Mb | 6.16 | 3.22 | 3.08 | 1.66 | 1.46 | 0.78 | 0.98 | 2.93 |
| | 10Mb | 61.5 | 32.2 | 30.8 | 16.6 | 14.6 | 7.81 | 9.77 | 29.3 |

The table shows that the cost of DEPSKY-A with $n = 4$ and without optimizations is roughly the sum of the costs of using the four clouds, as expected. However, if the read optimization is employed, the less expensive cloud cost dominates the cost of executing reads (only one out-of four clouds is accessed in fault-free executions). If the optimized write is employed, the data file will be written only on a preferred quorum excluding the most expensive cloud (Nirvanix), and thus the costs will be substantially smaller. For DEPSKY-CA, the cost of reading and writing without optimizations is approximately 50% of DEPSKY-A's due to the use of information-optimal erasure codes that make the data stored on each cloud roughly 50% of the size of the

original data. The optimized version of DEPSKY-CA also reduces the read cost to half of the sum of the two less costly clouds due to its access to only $f+1$ clouds in the best case, while the write cost is reduced since Nirvanix is not used. Recall that the costs for the optimized versions of the protocol account only for the best case in terms of monetary costs: reads and writes are executed on the required less expensive clouds. In the worst case, the more expensive clouds will also be used.

The storage costs of a 1Mb data unit for different numbers of stored versions is presented in Figure 4.8.1. We present the curves only for one data unit size because other size costs are directly proportional.




(a) Storage costs.

(b) DEPSKY-CA with preferred quorums.

Figure 4.5: Storage costs of a 1Mb data unit for different numbers of stored versions in different DEPSKY setups and clouds.

The results depicted in the Figure 4.5(a) show that the cost of DEPSKY-CA storage without employing preferred quorums is roughly half the cost of using DEPSKY-A and twice the cost of using a single cloud. This is no surprise since the storage costs are directly proportional to the amount of data stored on the cloud, and DEPSKY-A stores 4 times the data size, while DEPSKY-CA stores 2 times the data size and an individual cloud just stores a single copy of the data.

Figure 4.5(b) shows some results considering the case in which the data stored using DEPSKY-CA is stored only on a preferred quorum of clouds (see Section 4.3.7). The figure contains values for the less expensive preferred quorum (Amazon S3, Windows Azure and Rackspace) and the most expensive preferred quorum (Nirvanix, Windows Azure and Rackspace) together with Amazon S3 and DEPSKY-CA writing on all clouds for comparison. The results show that the use of preferred quorums decreases the storage costs between 15% (most expensive quorum) to 38% (less expensive quorum) when compared to the full replicated DEPSKY-CA. Moreover, in the best case, DEPSKY-CA can store data with an additional cost of only 23% more than the average cost to store data on a single cloud and twice the cost of the less expensive cloud (Amazon S3).

Notice that the metadata costs are almost irrelevant when compared with the data size since its size is less than 500 bytes.

### 4.8.2 Performance evaluation

In order to understand the performance of DEPSKY in a real deployment, we used PlanetLab to run clients accessing a cloud-of-clouds composed of popular storage cloud providers. This section explains our methodology and then presents the obtained results in terms of read and write latency, throughput and availability.

**Methodology.** The latency measurements were obtained using a logger application that tries to read a data unit from six different clouds: the four storage clouds individually and the two clouds-of-clouds implemented with DEPSKY-A and DEPSKY-CA.

The logger application executes periodically a *measurement epoch*, which comprises: read the data unit (DU) from each of the clouds individually, one after another; read the DU using DEPSKY-A; read the DU using DEPSKY-CA; sleep until the next epoch. The goal is to read the data through different setups within a time period as small as possible in order to minimize Internet performance variations.

We deployed the logger on eight PlanetLab machines across the Internet, on four continents. In each of these machines three instances of the logger were started for different DU sizes: 100kb (a measurement every 5 minutes), 1Mb (a measurement every 10 minutes) and 10Mb (a measurement every 30 minutes). These experiments took place during two months, but the values reported correspond to measurements done between September 10, 2010 and October 7, 2010.

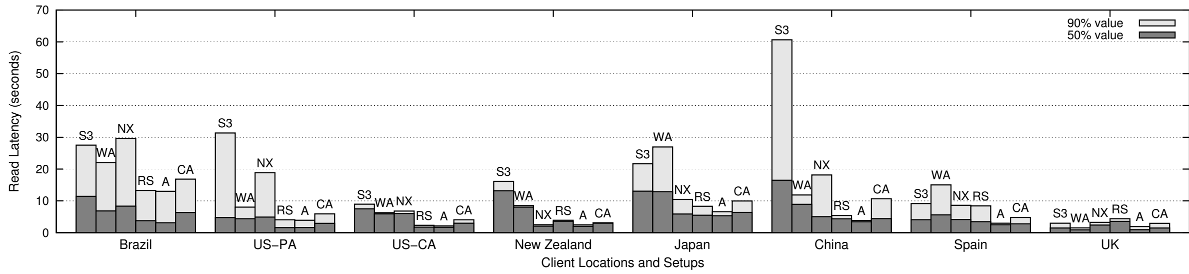
In the experiments, the local costs, in which the protocols incur due to the use of cryptography and erasure codes, are negligible for DEPSKY-A and account for at most 5% of the read and 10% of the write latencies on DEPSKY-CA.

**Reads.** Figure 4.8.2 presents the 50% and 90% percentile of all observed latencies of the reads executed (i.e., the values below which 50% and 90% of the observations fell). These experiments were executed without the (monetary) read optimization described in Section 4.3.7. The number of reads executed on each site is presented on the second column of Table 4.6.

Based on the results presented in the figure, several points can be highlighted. First, DEPSKY-A presents the best latency in all but one cases. This is explained by the fact that it waits for 3 out-of 4 copies of the metadata but only one of the data, and it usually obtains it from the best cloud available during the execution. Second, DEPSKY-CA's latency is closely related with the second best cloud storage provider, since it waits for at least 2 out-of 4 data blocks. Finally, there is a huge variance between the performance of the cloud providers when accessed from different parts of the world. This means that no provider covers all areas in the same way, and highlights another advantage of the cloud-of-clouds: we can adapt our accesses to use the best cloud for a certain location.

**The effect of optimizations.** An interesting observation of our DEPSKY-A (resp. DEPSKY-CA) read experiments is that in a significant percentage of the reads the cloud that replied metadata faster (resp. the two faster in replying metadata) is not the first to reply the data (resp. the two first in replying the data). More precisely, in 17% of the 60768 DEPSKY-A reads and 32% of the 60444 DEPSKY-CA reads we observed this behavior. A possible explanation for that could be that some clouds are better serving small files (DEPSKY metadata is around 500 bytes) and not so good on serving large files (like the 10Mb data unit of some experiments). This means that the read optimizations of Section 4.3.7 will make the protocol latency worse in these cases. Nonetheless we think this optimization is valuable since the rationale behind it

(a) 100kb DU.




(b) 1Mb DU.

(c) 10Mb DU.

Figure 4.6: $50^{th}/90^{th}$-percentile latency (in seconds) for 100kb, 1Mb and 10Mb DU read operations with PlanetLab clients located on different parts of the globe. The bar names are S3 for Amazon S3, WA for Windows Azure, NX for Nirvanix, RS for Rackspace, A for DEPSKY-A and CA for DEPSKY-CA. DEPSKY-CA and DEPSKY-A are configured with $n = 4$ and $f = 1$.

worked for more than 4/5 (DEPSKY-A) and 2/3 (DEPSKY-CA) of the reads in our experiments, and its use can decrease the monetary costs of executing a read by a quarter and half of the cost of the non-optimized protocol, respectively.

Table 4.4 shows, for each cloud (DEPSKY-A) or pair of clouds (DEPSKY-CA), the percentage of read operations that fetched data files from these clouds (i.e., these clouds answered first) for different client locations.

The first four lines of the table show that Rackspace was the cloud that provided the data file faster for most DEPSKY-A clients, while Amazon S3 provided the data more frequently for European clients. Interestingly, although these two clouds are consistently among the most used in operations coming from different parts of the world, it is difficult to decide between Windows Azure and Nirvanix to compose the preferred quorum to be used. Nirvanix showed to be fast for Asian clients (e.g., 45% of reads in Japan), while Windows Azure provided excellent performance in UK (e.g., 40% of reads fetched data from it). This tie can be broken considering the expected client location, the performance of writes and economical costs.

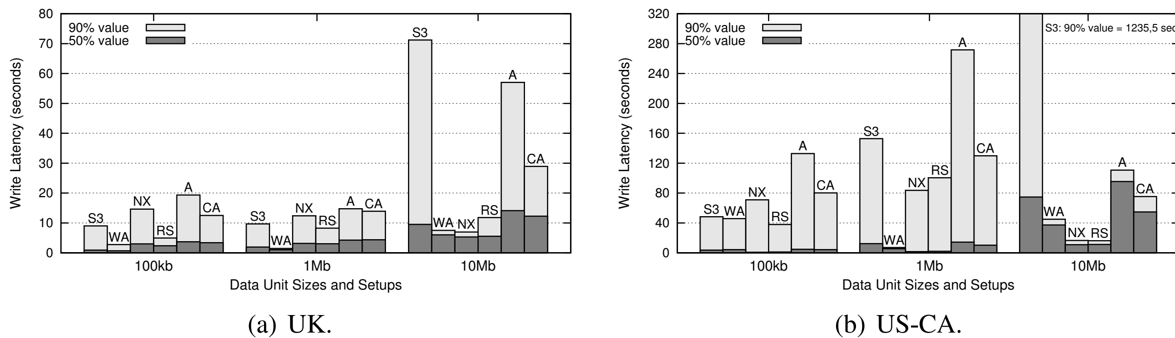
Considering DEPSKY-CA, where two data files are required to rebuild the original data,

Table 4.4: Percentage of reads in which the required data blocks were fetched from a specific cloud (or pair of clouds) for different locations. The clouds names are S3 for Amazon S3, WA for Windows Azure, NX for Nirvanix and RS for Rackspace. Results for single clouds stand for DEPSKY-A reads while results for a pair of clouds correspond to the 2 blocks read in DEPSKY-CA to rebuild the data.

| Cloud(s) | Brazil | US-PA | US-CA | New Z. | Japan | China | Spain | UK |
|---|---|---|---|---|---|---|---|---|
| S3 | 4 | 3 | 0 | 1 | 0 | 1 | 65 | 59 |
| NX | 0 | 2 | 0 | 14 | 45 | 2 | 2 | 0 |
| RS | 94 | 94 | 99 | 84 | 55 | 97 | 31 | 0 |
| WA | 1 | 1 | 0 | 0 | - | 0 | 2 | 40 |
| S3-RS | 53 | 61 | 2 | 3 | 1 | 3 | 67 | 2 |
| S3-NX | 0 | 1 | 0 | 0 | 0 | 1 | 3 | 0 |
| S3-WA | 0 | 1 | - | 0 | - | 0 | 2 | 81 |
| NX-WA | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 6 |
| NX-RS | 30 | 20 | 87 | 97 | 99 | 81 | 15 | 0 |
| RS-WA | 17 | 16 | 11 | 0 | 0 | 14 | 12 | 10 |

one can see that there are three possible preferred quorums for different locations: S3-RS-NX (Brazil, US-PA, New Zealand, Japan and Spain), NX-RS-WA (US-CA and China) and S3-WA-RS (UK). Again, the choice of the quorum used initially needs to be based on the other factors already mentioned. If one considers only the cost factor, the choice would be S3-RS-WA for both DEPSKY-A and DEPSKY-CA, since Windows Azure is much less expensive than Nirvanix (see Figure 4.5(b) in Section 4.8.1). By the other hand, as will be seen in the following, the perceived availability of Windows Azure was worse than Nirvanix in our experiments.

**Writes.** We modified our logger application to execute writes instead of reads and deployed it on the same machines we executed the reads. We run it for two days in October and collected the logs, with at least 500 measurements for each location and data size. These experiments were executed without the (monetary) read optimization described in Section 4.3.7. For the sake of brevity, we do not present all these results, but illustrate the costs of write operations for different data sizes and locations discussing only the observed results for UK and US-CA clients. The other locations present similar trends. These experiments were executed without the preferred quorum optimization described in Section 4.3.7. The 50% and 90% percentile of the latencies observed are presented in Figure 4.8.2.




(a) UK. (b) US-CA.

Figure 4.7: $50^{th}/90^{th}$-percentile latency (in seconds) for 100kb, 1Mb and 10Mb DU write operation for a PlanetLab client at the UK (a) and US-CA (b). The bar names are the same as in Figure 4.8.2. DEPSKY-A and DEPSKY-CA are configured with $n = 4$ and $f = 1$.

The latencies in the figure consider the time of writing the data on all four clouds (file sent

to 4 clouds, wait for only 3 confirmations) and the time of writing the new metadata. As can be observed in the figure, the latency of a write is of the same order of magnitude of a read of a DU of the same size (this was observed on all locations). It is interesting to observe that, while DEPSKY's read latency is close to the cloud with best latency, the write latency is close to the worst cloud. This comes from the fact that in a write DEPSKY needs to upload data blocks on all clouds, which consumes more bandwidth at the client side and requires replies from at least three clouds.

The figure also illustrates the big differences between the performance of the system depending on the client location. This difference is specially relevant when looking to the 90% values reported.

**Secret sharing overhead.** As discussed in Section 4.3.6, if a key distribution mechanism is available, secret sharing could be removed from DEPSKY-CA. However, the effect of this on read and write latencies would be negligible since *share* and *combine* (lines 9 and 34 of Algorithm 2) account for less than 3 and 0.5 ms, respectively. It means that secret sharing is responsible for less than 0.1% of the protocols latency in the worst case[3].

**Throughput.** Table 4.5 shows the throughput in the experiments for two locations: UK and US-CA. The values are of the throughput observed by a single client, not by multiple clients as done in some throughput experiments. The table shows read and write throughput for both DEPSKY-A and DEPSKY-CA, together with the values observed from Amazon S3, just to give a baseline. The results from other locations and clouds follow the same trends discussed here.

Table 4.5: Throughput observed in kb/s on all reads and writes executed for the case of 4 clouds ($f = 1$).

| Operation | DU Size | UK | | | US-CA | | |
|---|---|---|---|---|---|---|---|
| | | DEPSKY-A | DEPSKY-CA | Amazon S3 | DEPSKY-A | DEPSKY-CA | Amazon S3 |
| Read | 100kb | 189 | 135 | 59.3 | 129 | 64.9 | 31.5 |
| | 1Mb | 808 | 568 | 321 | 544 | 306 | 104 |
| | 10Mb | 1479 | 756 | 559 | 780 | 320 | 147 |
| Write | 100kb | 3.53 | 4.26 | 5.43 | 2.91 | 3.55 | 5.06 |
| | 1Mb | 14.9 | 26.2 | 53.1 | 13.6 | 19.9 | 25.5 |
| | 10Mb | 64.9 | 107 | 84.1 | 96.6 | 108 | 34.4 |

By the table it is possible to observe that the read throughput decreases from DEPSKY-A to DEPSKY-CA and then to Amazon S3, at the same time that write throughput increases for this same sequence. The higher read throughput of DEPSKY when compared with Amazon S3 is due to the fact that it fetches the data from all clouds on the same time, trying to obtain the data from the fastest cloud available. The price to pay for this benefit is the lower write throughput since data should be written at least on a quorum of clouds in order to complete a write. This trade off appears to be a good compromise since reads tend to dominate most workloads of storage systems.

The table also shows that increasing the size of the data unit improves throughput. Increasing the data unit size from 100kb to 1Mb improves the throughput by an average factor of 5 in both reads and writes. By the other hand, increasing the size from 1Mb to 10Mb shows less benefits: read throughput is increased only by an average factor of 1.5 while write throughput increases by an average factor of 3.3. These results show that cloud storage services should be

[3]For a more compreensive discussion about the overhead imposed by Java secret sharing see [BACF08].

used for storing large chunks of data. However, increasing the size of these chunks brings less benefit after a certain size (1Mb).

Notice that the observed throughputs are at least an order of magnitude lower than the throughput of disk access or replicated storage in a LAN [HGR07], but the elasticity of the cloud allows the throughput to grow indefinitely with the number of clients accessing the system (according to the cloud providers). This is actually the main reason that lead us to not trying to measure the peak throughput of services built on top of clouds. Another reason is that the Internet bandwidth would probably be the bottleneck of the throughput, not the clouds.

**Faults and availability.** During our experiments we observed a significant number of read operations on individual clouds that could not be completed due to some error. Table 4.6 presents the *perceived availability* of all setups calculated as $\frac{reads_completed}{reads_tried}$ from different locations.

Table 4.6: The perceived availability of all setups evaluated from different points of the Internet. The values were calculated as $\frac{reads_completed}{reads_tried}$.

| Location | Reads Tried | DepSky-A | DepSky-CA | Amazon S3 | Rackspace | Azure | Nirvanix |
|---|---|---|---|---|---|---|---|
| Brazil | 8428 | 1.0000 | 0.9998 | 1.0000 | 0.9997 | 0.9793 | 0.9986 |
| US-PA | 5113 | 1.0000 | 1.0000 | 0.9998 | 1.0000 | 1.0000 | 0.9880 |
| US-CA | 8084 | 1.0000 | 1.0000 | 0.9998 | 1.0000 | 1.0000 | 0.9996 |
| New Zealand | 8545 | 1.0000 | 1.0000 | 0.9998 | 1.0000 | 0.9542 | 0.9996 |
| Japan | 8392 | 1.0000 | 1.0000 | 0.9997 | 0.9998 | 0.9996 | 0.9997 |
| China | 8594 | 1.0000 | 1.0000 | 0.9997 | 1.0000 | 0.9994 | 1.0000 |
| Spain | 6550 | 1.0000 | 1.0000 | 1.0000 | 1.0000 | 0.9796 | 0.9995 |
| UK | 7069 | 1.0000 | 1.0000 | 0.9998 | 1.0000 | 1.0000 | 1.0000 |

The first thing that can be observed from the table is that the number of measurements taken from each location is not the same. This happens due to the natural unreliability of PlanetLab nodes, that crash and restart with some regularity.

There are two key observations that can be taken from Table 4.6. First, DepSky-A and DepSky-CA are the two single setups that presented an availability of 1.0000 in almost all locations[4]. Second, despite the fact that most cloud providers advertise providing 5 or 6 nines of availability, the perceived availability in our experiments was lower. The main problem is that outsourcing storage makes a company not only dependent on the provider's availability, but also on the network availability, which some studies show to have no more than two nines of availability [DCGN03]. This is a fact that companies moving critical applications to the cloud have to be fully aware.

## 4.9 Related Work

**Byzantine quorum systems.** DepSky provides a single-writer multi-reader read/write register abstraction built on a set of untrusted storage clouds that can fail in an arbitrary way. This type of abstraction supports an updatable data model, requiring protocols that can handle multiple versions of stored data. This is substantially different from providing write-once, read-maybe archival storages such as the one described in [SGMV07].

There are many protocols for Byzantine quorums systems for register implementation (e.g., [GWGR04,HGR07,MR97,MAD02]), however, few of them address the model in which servers

[4] This is somewhat surprising since we were expecting to have at least some faults on the client network that would disallow it to access any cloud.

are passive entities that do not run protocol code [ACKM06,AL03,JCT98]. DEPSKY differentiates from them in the following aspects: (1.) it decouples the write of timestamp and verification data from the write of the new value; (2.) it has optimal resiliency ($3f + 1$ servers [MAD02]) and employs read and write protocols requiring two communication round-trips independently of the existence of contention, faults and weakly consistent clouds; finally, (3.) it is the first single-writer multi-reader register implementation supporting efficient encoding and confidentiality. Regarding (2.), our protocols are similar to others for fail-prone shared memory (or "disk quorums"), where servers are passive disks that may crash or corrupt stored data. In particular, Byzantine disk Paxos [ACKM06] also presents a single-writer multi-reader regular register construction that requires two communication round-trips both for reading and writing in absence of contention. However, there is a fundamental difference between this construction and DEPSKY: it provides a weak liveness condition for the read protocol (termination only when there is a finite number of contending writes) while our protocol satisfies wait-freedom. An important consequence of this limitation is that reads may require several communication steps when contending writes are being executed. This same limitation appears on [AL03] that, additionally, does not tolerate writer faults. Regarding point (3.), it is worth to notice that several Byzantine storage protocols support efficient storage using erasure codes [CT06, GWGR04, HGR07], but none of them mention the use of secret sharing or the provision of confidentiality. However, it is not clear if information-efficient secret sharing [Kra93] or some variant of this technique could substitute the erasure codes employed on these protocols.

**Cloud storage availability.** Cloud storage is a hot topic with several papers appearing recently. However, most of these papers deal with the intricacies of implementing a storage infrastructure inside a cloud provider (e.g., [MJWS10]). Our work is closer to others that explore the use of existing cloud storage services to implement enriched storage applications. There are papers showing how to efficiently use storage clouds for file system backup [VSV09], implement a database [BFG+08], implement log-based file system [VSV12] or add provenance to the stored data [MRMS10]. However none of these works provide guarantees like confidentiality and availability and do not consider a cloud-of-clouds.

Some works on this trend deal with the high-availability of stored data through the replication of this data on several cloud providers, and thus are closely related with DEPSKY. The SafeStore system [KAD07a] provides an accountability layer for using a set of untrusted third-party storage systems in an efficient way. There are at least two features that make SafeStore very different from DEPSKY. First, it requires specific server-code on storage cloud provider (both in the service interface and in the internal storage nodes). Second, SafeStore does not support data sharing among clients (called SafeStore local servers) accessing the same storage services. The HAIL (High-Availability Integrity Layer) protocol set [BJO09] combines cryptographic protocols for proof of recoveries with erasure codes to provide a software layer to protect the integrity and availability of the stored data, even if the individual clouds are compromised by a malicious and mobile adversary. HAIL has at least three limitations when compared with DEPSKY: it only deals with static data (i.e., it is not possible to manage multiple versions of data), it requires that the servers run some code (opposite to DEPSKY, that uses the storage clouds as they are), and does not provide confidentiality guarantees for the stored data. The RACS (Redundant Array of Cloud Storage) system [ALPW10] employs RAID5-like techniques (mainly erasure codes) [PGK88] to implement high-available and storage-efficient data replication on diverse clouds. Differently from DEPSKY, RACS does not try to solve security problems of cloud storage, but instead deals with "economic failures" and vendor lock-in. In

consequence, the system does not provide any mechanism to detect and recover from data corruption or confidentiality violations. Moreover, it does not provide updates of the stored data. Finally, it is worth to mention that none of these cloud replication works present an experimental evaluation with diverse clouds as it is presented in this work.

**Cloud security.** There are several works about obtaining trustworthiness from untrusted clouds. Depot improves the resilience of cloud storage making similar assumptions to DepSky, that storage clouds are fault-prone black boxes [MSL+10]. However, it uses a single cloud, so it provides a solution that is cheaper but does not tolerate total data losses and the availability is constrained by the availability of the cloud on top of which it is implemented. Works like SPORC [FZFF10] and Venus [SCC+10] make similar assumptions to implement services on top of untrusted clouds. All these works consider a single cloud (not a cloud-of-clouds), require a cloud with the ability to run code, and have limited support for cloud unavailability, which makes them different from DepSky.

## 4.10 Conclusion

This chapter presents the design and evaluation of DepSky, a storage service that improves the availability and confidentiality provided by commercial storage cloud services. The system achieves these objectives by building a cloud-of-clouds on top of a set of storage clouds, combining Byzantine quorum system protocols, cryptographic secret sharing, erasure codes and the diversity provided by the use of several cloud providers. Beside of that, the notion of consistency proportionality introduced by DepSky allows the system to provide the same level of consistency of the underlying clouds it uses for storage.

We believe DepSky protocols are in an unexplored region of the quorum systems design space and can enable applications sharing critical data (e.g., financial, medical) to benefit from storage clouds. Moreover, the few and weak assumptions required by the protocols allow them to be used to replicate data efficiently not only on cloud storage services, but with any storage service available (e.g., NAS disks, NFS servers, FTP servers, key-value databases).

The chapter also presents an extensive evaluation of the system. The key conclusion is that it provides confidentiality and improved availability with an added cost as low as 23% more of the cost of storing data on a single cloud for a practical scenario, which seems to be a good compromise for critical applications.

In the next chapter we present a study of several properties of cloud-of-clouds object storage systems using distributed system theory tools.

# Chapter 5

# Object Storage: Theory

*Chapter Authors:*
Cristina Băsescu, Christian Cachin, Ittay Eyal, Robert Haas, Birgit Junker, Nikola Knežević, Alessandro Sorniotti and Marko Vukolić (IBM).

## 5.1 Overview

This chapter examines the theoretical foundations of distributed cloud storage services, such as DEPSKY (described in previous chapter). Such services often provide a *key-value store (KVS)* functionality, an object-based interface for accessing a collection of unstructured data items or *blobs*. Every blob is associated with a key that serves as identifier to access the blob.

In the first part, we present an efficient wait-free algorithm that emulates multi-reader multi-writer storage from a set of potentially faulty KVS replicas in an asynchronous environment. Our implementation serves an unbounded number of clients that use the storage concurrently. It tolerates crashes of a minority of the KVSs and crashes of any number of clients. Our algorithm minimizes the space overhead at the KVSs and comes in two variants providing regular and atomic semantics, respectively. Compared with prior solutions, it is inherently scalable and allows clients to write concurrently. Because of the limited interface of a KVS, textbook-style solutions for reliable storage either do not work or incur a prohibitively large storage overhead. Our algorithm maintains *two* copies of the stored value per KVS in the common case, and we show that this is indeed necessary. If there are concurrent write operations, the maximum space complexity of the algorithm grows in proportion to the point contention.

In the second part, we examine fundamental properties of the KVS abstraction. In the simplest form, a KVS provides only methods for writing and reading an entire blob, for removing blobs, and for listing all defined keys. On the other hand, many existing schemes for replicating data with the goal of enhancing resilience (e.g., based on quorum systems) associate logical *timestamps* with the stored values, in order to distinguish multiple versions of the same data item. This work uses the consensus number of a shared storage abstraction as a measure for its power to facilitate the implementation of data replication. It is demonstrated that a KVS is a very simple primitive, not different from read/write registers in this sense, and that a replica capable of the typical operations on timestamped data is fundamentally more powerful than a KVS. Hence, data replication schemes over storage providers with a KVS interface appear inherently more difficult to realize than replication schemes over providers with richer interfaces.

### 5.1.1 Motivation

In the recent years, the *key-value store* (*KVS*) abstraction has become the most popular way to access Internet-scale "cloud" storage systems. Such systems provide storage and coordination services for online platforms [DHJ+07, MTJ+08, ALM+10, LM10, Vol], ranging from web

search to social networks, but they are also available to consumers directly [Amad, CWO+11, racb, Mez].

A KVS offers a range of simple functions for manipulation of unstructured data objects, called *values*, each one identified by a unique *key*. While different services and systems offer various extensions to the KVS interface, the common denominator of existing KVS services implements an associative array: A client may *store* a value by associating the value with a key, *retrieve* a value associated with a key, *list* the keys that are currently associated, and *remove* a value associated with a key.

Although existing KVS services provide high availability and reliability using replication internally, a KVS service is managed by one provider; many common components (and thus failure modes) affect its operation. A problem with any such component may lead to service outage or even to data being lost, as witnessed during an Amazon S3 incident [Amab], Google's temporary loss of email data [Gma], and Amazon's recent service disruption [Amac]. As a remedy, a client may increase data reliability by replicating it among several storage providers (all offering a KVS interface), using the guarantees offered by *robust* distributed storage algorithms [Gif79, ABND95]. Data replication across different clouds is a topic of active research [ALPW10, CHV10, RP11, BCQ+11].

Replication over a multiple storage providers is not an easy task. KVS services provide an object-based interface for accessing a collection of unstructured *blobs*. Every blob is associated with a key that serves as identifier to access the blob. The common denominator of all current KVS services contains only methods for writing and reading an entire blob, for removing blobs, and for listing all defined keys. Exactly this simplistic interface poses a major problem for replication schemes that maintain versioned data on multiple KVS replicas. In particular, the first part of this chapter address this problem in detail and present a replication algorithm that maintains two copies of the stored value per KVS in the common case. There, we also show that storing two copies is necessary, in order to achieve wait-free client operations.

Data replication in the intercloud has recently received a lot of attention [BJO09, ALPW10, RP11, BCQ+11]. One class of such systems assume specialized interfaces on the storage replicas, which are capable of limited processing and command execution (like active disks [CM05] or the storage servers of HAIL [BJO09] and Cleversafe [RP11]). Replicas can therefore carry out limited computation, such as comparing timestamps and conditionally storing data. This feature is required by many replicated storage algorithms based on quorum systems, starting with some of the first schemes [CBPS10, ABND95, CGR11]. A second class of cloud-data replication schemes, in particular RACS [ALPW10] and DepSky [BCQ+11], uses a KVS provider; they compensate for the relative simplicity of the KVS interface by adding extra components for synchronization among multiple clients.

In the second part of this chapter, we identify an inherent difference between the storage abstractions used by the two classes of replication systems mentioned before. We examine the power of the popular KVS model from the perspective of the designer of a failure- and intrusion-tolerant replication scheme. A replication method enables multiple clients to operate on a storage abstraction emulated from a pool of potentially faulty storage providers, such as cloud-based KVSs; implicitly, the richness, complexity, and performance of the emulated service depends on the power of the underlying primitives. We analyze the *consensus number* storage abstraction as a measure for its capability to provide wait-free synchronization among the set of clients according to Herlihy's fundamental notion [Her91, HS08].

### 5.1.2 Contribution

First, we provide the theoretical formalism for building robust data sharing over replicated sets of key-value stores (KVS). Toward that end, we present a robust, asynchronous, and space-efficient emulation of a register over a set of KVSs, which may fail by crashing. Our formalization of a key-value store object represents the common denominator among existing commercial KVSs, which renders our approach feasible in practice. Inspired by Internet-scale systems, the emulation is designed for an unbounded number of clients and supports multiple readers and writers (MRMW). The algorithm is *wait-free* [HW90] in the sense that all operations invoked by a correct client eventually complete. It is also *optimally resilient*, i.e., tolerates the failure of any minority of the KVSs and of any number of clients.

Next, we present two variations of the emulation. A basic algorithm emulates a register with *regular* semantics in the multi-writer model [SPW03]. It does not require read operations to write to the KVSs. Precluding readers from writing is practically appealing, since the clients may belong to different domains and not all readers may have write privileges for the shared memory. But it also poses a challenge because of the garbage-collection (GC) racing problem. Our solution stores the same value *twice* in every KVS: (1) under an *eternal* key, which is never removed by a garbage collector, and therefore is vulnerable to an old-new overwrite and (2) under a *temporary* key, named according to the version; obsolete temporary keys are garbage-collected by write operations, which makes these keys vulnerable to the GC racing problem. The algorithm for reading accesses the values in the KVSs according to a specific order, which guarantees that every read terminates eventually despite concurrent write operations. In a sense, the eternal and temporary copies complement each other and, together, guarantee the desirable properties of this emulation.

We then present an extension to the basic algorithm, that enables the emulation of an *atomic* register [Lam86]. It uses the standard approach of having the readers write back the returned value [ABND95]. This algorithm requires read operations to write, but this is necessary [Lam86, AW04].

These two emulations emulations maintain only two copies of the stored value per KVS in the common case (i.e., failure-free executions without concurrent operations). We show that this is also necessary. In the worst case, a stored value exists in every KVS once for every concurrent write operation, in addition to the one stored under the eternal key. Hence, these emulations have optimal space complexity.

Even though it is well-known how to implement a shared, robust multi-writer register from simpler storage primitives such as unreliable single-writer registers [AW04], the presented algorithm is the first to achieve an emulation from KVSs with the minimum necessary space overhead.

Furthermore, we note that some of the available KVSs export proprietary versioning information [Amad, Vol]. However, one cannot exploit this for a data replication algorithm before the format and semantics of those versions has been harmonized. Another KVS prototype allows to execute client operations [GLK+10], but this technique is far from commercial deployment. We believe that some KVSs may also support atomic "read-modify-write" operations at some future time, thereby eliminating the problem addressed here. But until these extensions are deployed widely and have been standardized, our algorithm represents the best possible solution for minimizing space overhead of data replication on KVSs.

Lastly, in the second part of this chapter, we focus on the workings of a basic building block of our replication algorithm — a single KVS. There, we show that the typical processing steps expected from a replica in traditional replicated storage schemes give it universal power —

these replicas have infinite consensus number and are as powerful as the consensus abstraction for implementing other concurrent data structures. A key-value store, on the other hand, has the least amount of synchronization power available in any shared object that has been studied — the consensus number of a KVS is one and falls into the same class as a simple read/write register. These results illustrate why data replication for typical cloud storage systems requires additional mechanisms for letting multiple clients access the system concurrently.

## 5.2 Model

Here we introduce the formal model underlying the description of algorithms in this chapter. Next, we define linearizability and wait-freedom, followed by specifications of registers with regular and atomic semantics. Then, we introduce the consensus number as a measure for the synchronization power of a shared object. Finally, we introduce a key-value store object and state the system model.

### 5.2.1 Executions

The system is comprised of unbounded number of *clients* and *(base) objects*. We model them as I/O automata [Lyn96], which contain state and potential transitions that are triggered by *actions*. The interface of an I/O automaton is determined by external (input and output) actions. A client may *invoke* an *operation*[1] on an object (with an output action of the client automaton that is also an input action of the object automaton). The object reacts to this invocation, possibly involving state transitions and internal actions, and returns a *response* (an output action of the object that is also an input action of the client). This *completes* the operation. In other words, each operation consists of two events, the *invocation* and the *response*.

We consider an asynchronous system, i.e., there are no timing assumptions that relate invocations and responses. (Consult [Lyn96, AW04] for details.)

Clients and objects may *fail* by stopping, i.e., *crashing*, which we model by a special action *stop*. When *stop* occurs at automaton $A$, all actions of $A$ become disabled indefinitely and $A$ no longer modifies its state. A client or base object that does not fail is called *correct*.

### 5.2.2 Linearizability

The sequence of invocations and responses of $O$ occurring in an execution $\sigma$ are called a *history*. An invocation and a response *match* if they are both events occurring at the same client, concern the same object, and the response occurs after the invocation and before any other invocation concerning the same object. An operation whose invocation appears in a history is *complete* if the history also contains a matching response. The subsequence of $\sigma$ containing only the complete operations of $\sigma$ is denoted by $complete(\sigma)$. If an operation is not complete in a history it is called *pending*. An *extension* of $\sigma$ is any history that can be obtained from $\sigma$ by appending responses for any subset of the pending requests in $\sigma$.

In a sequence of events $\sigma$ an operation $o$ *precedes* another operation $o'$ if $o$ completes before $o'$ is invoked. This is denoted by $o <_\sigma o'$. If neither of two operations precedes the other, they are *concurrent*. A sequence of events that does not contain any concurrent events is called *sequential*. We assume that every client invokes operations on one object in a *well-formed way*,

[1] For simplicity, we refer to an *operation* when we should be referring to *operation execution*.

that is, the client never invokes an operation on an object when an operation by that client on the same object is pending.

A history $\pi$ consisting only of events in a history $\sigma$ is said to *preserve the real-time order* of $\sigma$ if for any two operations $o$ and $o'$ in $\pi$, the condition $o <_\sigma o'$ implies that $o <_\pi o'$.

A *client subhistory* of $\sigma$ for a client $c$ is the subsequence of $\sigma$ that contains only those events that occur at $c$; it is denoted by $\sigma|_c$. For an object $O$, the *object subhistory* $\sigma|_O$ is defined analogously. Two histories $\sigma$ and $\sigma'$ are *equivalent* if for every client $c$ it holds that $\sigma|_c = \sigma'|_c$.

The *sequential specification* defines a shared object by describing its behavior in sequential executions. A history $\sigma$ is *legal* if each of its object subhistories is legal with respect to the sequential specification.

An important class of objects appear to execute operations "atomically," as captured by the notion of *linearizability* formalized by Herlihy and Wing [HW90]. We consider only linearizable semantics in this work.

**Definition 1** (Linearizability)**.** *A history of events $\sigma$ is* linearizable *if it has an extension $\sigma'$ and there exists a legal sequential history $\pi$ such that:*

1. *complete$(\sigma')$ is equivalent to $\pi$; and*
2. *$\pi$ preserves the real-time order of $\sigma$.*

### 5.2.3 Wait-freedom

In a system where several clients execute operations on shared object(s), none of the clients should be prevented from making progress due to operations of other clients. A system achieving this property is called *wait-free*. This is an important aspect of resilient Internet services. The notion was made formal by Herlihy [Her91] but probably appears first in the work of Lamport [Lam74]. We consider only wait-free systems in the remainder of this work.

**Definition 2** (Wait-Freedom)**.** *Consider a system where several clients access a shared object. The system is* wait-free *if in all executions, every client gets a response to an operation invocation within a finite number of steps, that is, independent of failures and of actions of the other clients.*

### 5.2.4 Register Specifications

**Sequential Register.** A *register* [Lam86] is an object that supports two operations: one for writing a value $v \in \mathcal{V}$, denoted by **write**$(v)$, which returns ACK, and one for reading a value, denoted by **read**$()$, which returns a value in $\mathcal{V}$. The sequential specification of a register requires that every **read** operation returns the value written by the last preceding **write** operation in the execution, or the special value $\bot$ if no such operation exists. For simplicity, our description assumes that every distinct value is written only once.

Registers may exhibit different semantics under concurrent access, as described next.

**Multi-Reader Multi-Writer Regular Register.** The following semantics describe a *multi-reader multi-writer regular register* (*MRMW-regular*), adapted from [SPW03]. A MRMW-regular register only guarantees that different **read** operations agree on the order of preceding **write** operations.

**Definition 3** (MRMW-regular register)**.** *A well-formed execution $\sigma$ of a register is MRMW-regular if there exists a sequential permutation $\pi$ of the operations in $\sigma$ as follows: for each* ***read*** *operation $r$ in $\sigma$, let $\pi_r$ be a subsequence of $\pi$ containing $r$ and those* ***write*** *operations that do not follow $r$ in $\sigma$; furthermore, let $\sigma_r$ be the subsequence of $\sigma$ containing $r$ and those* ***write*** *operations that do not follow it in $\sigma$; then $\pi_r$ is a legal real-time sequential permutation of $\sigma_r$. A register is MRMW-regular if all well-formed executions on that register are MRMW-regular.*

**Atomic Register.** A stronger consistency notion for a concurrent register object than regular semantics is *atomicity* [Lam86], also called linearizability [HW90]. In short, atomicity stipulates that it should be possible to place each operation at a singular point (linearization point) between its invocation and response.

**Definition 4** (Atomicity)**.** *A well-formed execution $\sigma$ of a concurrent object is* atomic *(or* linearizable*), if $\sigma$ can be extended (by appending zero or more responses) to some execution $\sigma'$, such that there is a legal real-time sequential permutation $\pi$ of $\sigma'$. An object is atomic if all well-formed executions on that object are atomic.*

### 5.2.5 Key-Value Store

A *key-value store* (*KVS*) object is an associative array that allows storage and retrieval of *values* in a set $\mathcal{X}$ associated with *keys* in a set $\mathcal{K}$. The size of the stored values is typically much larger than the length of a key, so the values in $\mathcal{X}$ cannot be translated to elements of $\mathcal{K}$ and be stored as keys.

A KVS supports four operations: (1) *Storing* a value $x$ associated with a key *key* (denoted $\mathbf{put}(key, x)$), (2) *retrieving* a value $x$ associated with a key ($x \leftarrow \mathbf{get}(key)$), which may also return FAIL if *key* does not exist, (3) *listing* the keys that are currently associated ($list \leftarrow \mathbf{list}()$), and (4) *removing* a value associated with a key ($\mathbf{remove}(key)$).

Our formal sequential specification of the KVS object is given in Algorithm 4. This implementation maintains in a variable *live* the set of associated keys and values. The *space complexity* of a KVS at some time during an execution is given by the number of associated keys, that is, by the value $|live|$.

### 5.2.6 System model

The system is comprised of a finite set of clients and a set of $n$ atomic wait-free KVSs as base objects. Each client is named with a unique identifier from an infinite ordered set $\mathcal{ID}$. The KVS objects are numbered $1, \ldots, n$. Initially, the clients do not know the identities of other clients or the total number of clients.

Our goal is to have the clients *emulate* a MRMW-regular register and an atomic register using the KVS base objects [Lyn96]. The emulations should be wait-free and tolerate that any number of clients and any minority of the KVSs may crash. Furthermore, an emulation algorithm should associate only few keys to values in every KVS (i.e., have low space complexity).

### 5.2.7 Consensus number

The *consensus problem* requires multiple clients to agree on a common value from a set of proposed values. A *consensus object* abstracts a service that provides a wait-free implementation of a distributed protocol that solves the consensus problem [Her91].

**Algorithm 4:** Key-value store object $i$

```
1  state
2      live ⊆ 𝒦 × 𝒳, initially ∅
3  On invocation put_i(key, value)
4      live ← (live \ {⟨key, x⟩ | x ∈ 𝒳}) ∪ ⟨key, value⟩
5      return ACK
6  On invocation get_i(key)
7      if ∃x : ⟨key, x⟩ ∈ live then
8          return x
9      else
10         return FAIL
11 On invocation remove_i(key)
12     live ← live \ {⟨key, x⟩ | x ∈ 𝒳}
13     return ACK
14 On invocation list_i()
15     return {key | ∃x : ⟨key, x⟩ ∈ live}
```

**Definition 5** (Consensus Object)**.** *A* consensus object *is a shared object with one operation* $decide(v)$*; it takes a value* $v$*, called a* proposal*, as input parameter and returns a* decision value*. Every client calls decide with its own proposal* $v$ *at most once. The returned decision value* $d$ *satisfies:*

1. (Validity) *The value* $d$ *is the proposal of some client; and*
2. (Consistency) *All clients return the same decision value* $d$*.*

A consensus object permits any number of clients. For simplicity we consider only *binary consensus* in this work, where the proposals are either zero or one.

Next we introduce the concept of consensus numbers, which is an important measure for the synchronization power of a shared object in a wait-free system.

**Definition 6** (Consensus Number [Her91])**.** *The* consensus number *of a shared object is the maximum number of clients for which this object can solve the consensus problem. If no maximum exists, the consensus number is said to be infinite.*

The consensus number provides a measure for classifying shared objects with respect to their power to synchronize multiple concurrent clients. By definition, a consensus object has infinite consensus number; consensus has been called *universal* for this reason. One of the simplest objects considered in the literature is a read/write register; it has consensus number one [HS08]. The following result, first shown by Herlihy [Her91], organizes all shared objects in a hierarchy based on their consensus numbers.

**Theorem 1** ( [Her91])**.** *If an object* $X$ *has consensus number* $n$ *and another object* $Y$ *has consensus number* $m < n$*, then* $X$ *cannot be implemented in a wait-free way from* $Y$ *in a system with more than* $m$ *clients.*

In other words, a given shared object is strictly more powerful than any other shared object that has a smaller consensus number.

## 5.3 On Robust Data Sharing with Key-Value Stores

This section describes the theoretical formalism of enhancing the dependability of KVS services through replication over multiple clouds.

Our data replication scheme relies on multiple providers of raw storage, called *base objects* here, and emulates a single, more reliable shared storage abstraction, which we model as a *read/write register*. A register represents the most basic form of storage, from which a KVS service or more elaborate abstractions may be constructed. The emulated register tolerates asynchrony, concurrency, and faults among the clients and the base objects. For increased parallelism, the clients do not communicate with each other for coordination, and they may not even be aware of each other.

Many well-known robust distributed storage algorithms exist (for an overview see [CGR11]). They all use versioning [VA86], whereby each stored value is associated with a logical timestamp. For instance, with the multi-writer variant of the register emulation by Attiya et al. [ABND95], the base objects perform custom *computation* depending on the timestamp, in order to identify and to retain only the newest written value. Without this an *old-new overwrite* problem might occur when a slow write request with an old value and a small timestamp reaches a base object after the latter has already updated its state to a newer value with a higher timestamp. On the other hand, one might let each client use its own range of timestamps and retain all versions of a written value at the KVSs [GL03, ACKM06], but this approach is overly expensive in the sense that it requires as many base objects as there are clients. If periodic garbage collection (GC) is introduced to reduce the consumed storage space, one may face a *GC racing* problem, whereby a client attempts to retrieve a value associated with a key that has become obsolete and was removed.

### 5.3.1 Algorithm

#### Pseudo Code Notation

Our algorithm is formulated using functions that execute the register operations. They perform computation steps, invoke operations on the base objects, and may *wait for* such operations to complete. To simplify the pseudo code, we imagine there are concurrent execution "threads" as follows. When a function **concurrently** executes a block, it performs the same steps and invokes the same operations once for each KVS base object in parallel. An algorithm proceeds past a **concurrently** statement as indicated by a termination property; in all our algorithms, this condition requires that the block completes for a majority of base objects.

In order to maintain a well-formed execution, the system implicitly keeps track of pending operations at the base objects. Relying on this state, every instruction to **concurrently** execute a code block explicitly waits for a base object to complete a pending operation, before its "thread" may invoke another operation. This convention avoids cluttering the pseudo code with state variables and complicated predicates that have the same effect.

#### MRMW-Regular Register

We present an algorithm for implementing a MRMW-regular register, where **read** operations do not store data at the KVSs.

Inspired by previous work on fault-tolerant register emulations, our algorithm makes use of versioning. Clients associate versions with the values they store in the KVSs. In each KVS there

may be several values stored at any time, with different versions. Roughly speaking, when writing a value, a client associates it with a version that is larger than the existing versions, and when reading a value, a client tries to retrieve the one associated with the largest version [ABND95]. Since a KVS cannot perform computations and atomically store one version and remove another one, values associated with obsolete versions may be left around. Therefore our algorithm explicitly removes unused values, in order to reduce the space occupied at a KVS.

A version is a pair[2] $\langle seq, id \rangle \in \mathbb{N}_0 \times \mathcal{ID}$, where the first number is a sequence number and the second is the identity of the client that created the version and used it to store a value. When comparing versions with the $<$ operator and using the $\max$ function, we respect the lexicographic order on pairs. We assume that the key space of a KVS is the version space, i.e., $\mathcal{K} = \mathbb{N}_0 \times \mathcal{ID}$, and that the value space of a KVS allows clients to store either a register value from $\mathcal{V}$ or a version and a value in $(\mathbb{N}_0 \times \mathcal{ID}) \times \mathcal{V}$.[3]

At the heart of our algorithm lies the idea of using *temporary keys*, which are created and later removed at the KVSs, and an *eternal key*, denoted ETERNAL, which is never removed. Both represent a register value and its associated version. When a client writes a value to the emulated register, it determines the new version to be associated with the value, accesses a majority of the KVSs, and stores the value and version *twice* at every KVS — once under a new temporary key, named according to the version, and once under the eternal key, overwriting its current value. The data stored under a temporary key directly represents the written value; data stored under the eternal key contains the register value and its version. The writer also performs garbage collection of values stored under obsolete temporary keys, which ensures the bound on space complexity.

**Read** When a client reads from the emulated register through algorithm **regularRead** (Algorithm 6), it obtains a version and a value from a majority of the KVSs and returns the value associated with the largest obtained version.

To obtain such a pair from a KVS $i$, the reader invokes a function **getFromKVS**$(i)$ (shown in Algorithm 5). It first determines the currently largest stored version, denoted by $ver_0$, through a snapshot of temporary keys with a **list** operation.

Then the reader enters a loop, from which it only exits after finding a value associated with a version that is at least $ver_0$. It first attempts to retrieve the value under the key representing the largest version. If the key exists, the reader has found a suitable value. However, this step may fail due to the GC racing problem, that is, because a concurrent writer has removed the particular key between the times when the client issues the **list** and the **get** operations.

In this case, the reader retrieves the version/value pair stored under the eternal key. As the eternal key is stored first by a writer and never removed, it exists always after the first write to the register. If the retrieved version is greater than or equal to $ver_0$, the reader returns this value. However, if this version is smaller than $ver_0$, an old-new overwrite has occurred, and the reader starts another iteration of the loop.

This loop terminates after a bounded number of iterations: Note that an iteration is not successful only if a GC race and an old-new overwrite have both occurred. But a concurrent writer that may cause an old-new overwrite must have invoked its write operation *before* the reader issued the first **list** operation on some KVS. Thus, the number of loop iterations is bounded by the

[2] We denote by $\mathbb{N}_0$ the set $\{0, 1, 2, \dots\}$.

[3] In other words, $\mathcal{X} = \mathcal{V} \cup (\mathbb{N}_0 \times \mathcal{ID}) \times \mathcal{V}$. Alternatively one may assume that there exists a one-to-one transformation from the version space to the KVS key space, and from the set of values written by the clients to the KVS value space. In practical systems, where $\mathcal{K}$ and $\mathcal{X}$ are strings, this assumptions holds.

number of clients that concurrently execute a **write** operation in parallel to the **read** operation (i.e., the point contention of **write** operations). This intuition is made formal in Section B.1.

**Algorithm 5:** Retrieve a legal version-value pair from a KVS

```
1  function getFromKVS(i)
2      list ← list_i() \ ETERNAL
3      if list = ∅ then
4          return ⟨⟨0, ⊥⟩, ⊥⟩
5      ver_0 ← max(list)
6      while True do
7          val ← get_i(max(list))
8          if val ≠ FAIL then
9              return ⟨max(list), val⟩
10         ⟨ver, val⟩ ← get_i(ETERNAL)
11         if ver ≥ ver_0 then
12             return ⟨ver, val⟩
13         list ← list_i() \ ETERNAL
```

**Algorithm 6:** Client $c$ **read** operation of the MRMW-regular register

```
1  function regularRead_c()
2      results ← ∅
3      concurrently for each 1 ≤ i ≤ n, until a majority completes
4          if some operation is pending at KVS i then wait for a response
5          result ← getFromKVS(i)
6          results ← results ∪ {result}
7      return val such that ⟨ver, val⟩ ∈ results and ver′ ≤ ver for any ⟨ver′, val′⟩ ∈ results
```

**Write** A client writes a value to the register using algorithm **regularWrite** (Algorithm 8). First, the client lists the temporary keys in each base object and determines the largest version found in a majority of them. It increments this version and obtains a new version to be associated with the written value.

Then the client stores the value and the new version in all KVSs using a function **putInKVS**, shown in Algorithm 7, which also performs garbage collection. It first lists the existing keys and removes obsolete temporary keys, i.e., all temporary keys excluding the one corresponding to the maximal version. Subsequently the function stores the value and the version under the eternal key. To store the value under a temporary key, the algorithm checks whether the new version is larger than the maximal version of an existing key. If yes, it also stores the new value under the temporary key corresponding to the new version and removes the key holding the previous maximal version.

Once the function **putInKVS** finishes for a majority of the KVSs, the algorithm for writing to the register completes. It is important for ensuring termination of concurrent **read** operations that the writer first stores the value under the eternal key and later under the temporary key.

**Algorithm 7:** Store a value and a given version in a KVS

1 **function putInKVS**$(i, ver_w, val_w)$
2 $\quad$ $list \leftarrow$ **list**$_i()$
3 $\quad$ $obsolete \leftarrow \{v \mid v \in list \wedge v \neq \text{ETERNAL} \wedge v < \max(list)\}$
4 $\quad$ **foreach** $ver \in obsolete$ **do**
5 $\quad\quad$ **remove**$_i(ver)$
6 $\quad$ **put**$_i(\text{ETERNAL}, \langle ver_w, val_w \rangle)$
7 $\quad$ **if** $ver_w > \max(list)$ **then**
8 $\quad\quad$ **put**$_i(ver_w, val_w)$
9 $\quad\quad$ **remove**$_i(\max(list))$

**Algorithm 8:** Client $c$ **write** operation of the MRMW-regular register

1 **function regularWrite**$_c(val_w)$
2 $\quad$ $results \leftarrow \{\langle 0, \bot \rangle\}$
3 $\quad$ **concurrently** for each $1 \leq i \leq n$, until a majority completes
4 $\quad\quad$ **if** some operation is pending at KVS $i$ **then** wait for a response
5 $\quad\quad$ $list \leftarrow$ **list**$_i()$
6 $\quad\quad$ $results \leftarrow results \cup list$
7 $\quad$ $\langle seq_{\max}, id_{\max} \rangle \leftarrow \max(results)$
8 $\quad$ $ver_w \leftarrow \langle seq_{\max} + 1, c \rangle$
9 $\quad$ **concurrently** for each $1 \leq i \leq n$, until a majority completes
10 $\quad\quad$ **if** some operation is pending at KVS $i$ **then** wait for a response
11 $\quad\quad$ **putInKVS**$(i, ver_w, val_w)$
12 $\quad$ **return** ACK

#### Atomic Register

The atomic register emulation results from extending the algorithm for emulating the regular register. Atomicity is achieved by having a client write back its read value before returning it, similar to the write-back procedure of Attiya et al. [ABND95].

The **write** operation is the same as before, implemented by function **regularWrite** (Algorithm 8). The **read** operation is implemented by function **atomicRead** (Algorithm 9). Its first phase is unchanged from before and obtains the value associated with the maximal version found among a majority of the KVSs. Its second phase duplicates the second phase of the **regularWrite** function, which stores the versioned value to a majority of the KVSs.

### 5.3.2 Correctness

For the sake of brevity, we present the correctness proofs for our regular and atomic register constructions in a clearly marked appendix. We suggest the interested reader to consult Appendix B.

### 5.3.3 Efficiency

We discuss the space complexity of the algorithms in this section. Our algorithms emulate a MRMW-regular and atomic registers from KVS base objects. The standard emulations of

**Algorithm 9:** Client $c$ **read** operation of the atomic register

```
1  function atomicRead_c()
2      results ← ∅
3      concurrently for each 1 ≤ i ≤ n, until a majority completes
4          if some operation is pending at KVS i then wait for a response
5          result ← getFromKVS(i)
6          results ← results ∪ {result}
7      choose ⟨ver, val⟩ ∈ results such that ver′ ≤ ver for any ⟨ver′, val′⟩ ∈ results
8      concurrently for each 1 ≤ i ≤ n, until a majority completes
9          if some operation is pending at KVS i then wait for a response
10         putInKVS(i, ver, val)
11     return val
```

such registers use base objects with atomic read-modify-write semantics, which may receive versioned values and always retain the value with the largest version. Since a KVS has simpler semantics, our emulations store more than one value in each KVS.

Note how the algorithm for writing performs garbage collection on a KVS *before* storing a temporary key in the KVS. This is actually necessary for bounding the space at the KVS, since the **putInKVS** function is called concurrently for all KVSs and may be aborted for some of them. If the algorithm would remove the obsolete temporary keys *after* storing the value, the function may be aborted just before garbage collection. In this way, many obsolete keys might be left around and permantly occupy space at the KVS.

We provide upper bounds on the space usage in Section 5.3.3 and continue in Section 5.3.3 with a lower bound. The time complexity of our emulations follows from analogous arguments.

#### Maximal Space Complexity

It is obvious from Algorithm 8 that when a **write** operation runs in isolation (i.e., without any concurrent operations) and completes the **putInKVS** function on a set $\mathcal{C}$ of more than $n/2$ correct KVSs, then every KVS in $\mathcal{C}$ stores only the eternal key and one temporary key. Every such KVS has space complexity two. When there are concurrent operations, the space complexity may increase by one for every concurrent write operation. Recall that point contention denotes the maximal number of clients executing an operation concurrently.

**Theorem 2.** *The space complexity of the MRMW-regular register emulation at any KVS is at most two plus the point contention of concurrent write operations.*

The proof of this theorem is presented in Section B.2 of the appendix.

A similar theorem holds for the atomic register emulation, except here **read** operations may also increase the space complexity. The proof is similar to that of the regular register, and is omitted for brevity.

**Theorem 3.** *For any execution $\bar{\sigma}$, the maximal storage occupied by the atomic algorithm on a KVS $i$ is at most linear in the concurrent number of operations.*

#### Minimal Space Complexity

We show that every emulation of even a *safe* [Lam78] register, which is weaker than a regular register, from KVS base objects incurs space complexity two at the KVS objects.

**Algorithm 1** Sequential spec. of the replica object $R$

**state**
  $(R.ts, R.v)$, initially $(0, \perp)$;

**operation** *condwrite*$(ts, v)$
  **if** $ts > R.ts$ **then**
    $(R.ts, R.v) \leftarrow (ts, v)$;
  **return** ACK;

**operation** *read*$()$
  **return** $R.v$;

**Theorem 4.** *In every emulation of a safe MRMW-register from KVS base objects, there exists some KVS with space complexity two.*

The proof of this theorem is presented in Section B.2 of the appendix.

## 5.4 On Limitations of Using Cloud Storage for Data Replication

Many protocols that implement robust shared memory in distributed systems use the notion of logical timestamps [Lam78] for identifying different versions of a stored value over time. They usually maintain the stored value in the form of a pair, consisting of a timestamp $ts$ and the actual value $v$.

We now introduce a *replica object*, which is inherent in a large number of distributed implementations of shared memory; it corresponds, for example, to the processors used by Attiya et al. [ABND95] and to the active disks of Chockler and Malkhi [CM05]. Our replica object provides functionality to conditionally store a timestamp/value pair, which is required from the storage primitive in many robust shared storage implementations. It serves any number of clients.

More precisely, a replica object $R$ stores a timestamp/value pair internally and offers two operations, called *condwrite* and *read*, as shown in Algorithm 1. Operation *condwrite*$(ts, v)$ takes a timestamp/value pair as input and returns a constant symbol; it only stores the value $v$ in the replica object if the timestamp $ts$ is bigger than the internally stored timestamp. Operation *read* takes no input and returns a value; it simply accesses the internally stored value and returns it.

From the point of view of synchronization, replica objects can be used to implement a consensus object for any number of clients. Hence, a replica is universal and can implement any synchronization primitive.

**Theorem 5.** *The consensus number of a replica object is infinite.*

The proof of this theorem is presented in Section B.3 of the appendix.

### 5.4.1 The consensus number of a key-value store

A key-value store (KVS) represents an object-based storage service, which has become popular in the context of cloud storage. Pioneered by Amazon S3 [Amad], it now represents a de-

facto standard for many commercial cloud storage services (e.g., Windows Azure [CWO+11], Rackspace [racb], and many others [jcl]).

This section illustrates the fundamental power of a KVS for wait-free synchronization. We show how to implement a KVS from a so-called snapshot object in a wait-free manner. Snapshot objects represent a prominent abstraction of shared storage with many applications. Since a snapshot object can be implemented from register objects, we can show that the KVS object has consensus number one.

### Snapshot objects

A *snapshot object* [AAD+93], abbreviated *SO*, is a shared object that stores $n$ values in a system of $n$ clients, one value per client. In a single-writer snapshot object, as considered here, every value may only be written by the corresponding client and all clients may read all values.

More precisely, an atomic snapshot object $SO$ maintains a vector $D$ of $n$ values from a domain $\mathcal{V}$ and provides two operations, denoted *update* and *scan*. When a client with an index $i \in \{1, \ldots, n\}$ invokes *update*$(i, v)$ for a value $v \in \mathcal{V}$, then $SO$ atomically sets $D[i] \leftarrow v$ and responds with an acknowledgment. No client may invoke *update* with the index of another client. Operation *scan*$()$ with no parameters may be invoked by any client and returns the vector $D$.

The sequential specification of an atomic snapshot object requires that for each $D$ returned by *scan*, entry $D[i]$ for $i = 1, \ldots, n$ equals the value $d$ given in the most recent preceding *update*$(i, d)$ operation by the client with index $i$; if there is no such preceding *update*, then $D[i]$ is equal to $\perp$.

Interestingly, one can implement an atomic snapshot object for any number of clients only from atomic read/write registers [AAD+93]. Because registers have consensus number one, atomic snapshot objects have consensus number one.

### From snapshot objects to KVS objects

This section gives a constructive proof of the following theorem, by exhibiting a wait-free implementation of a KVS object from atomic snapshot objects.

**Theorem 6.** *The consensus number of a key-value store object is one.*

The proof of this theorem is presented in Section B.3 of the appendix.

## 5.5 Related Work

There is a rich body of literature on robust register emulations that provide guarantees similar to ours. However, virtually all of them assume read-modify-write functionalities, that is, they rely on atomic computation steps at the base objects. These include the single-writer multi-reader (SWMR) atomic wait-free register implementation of Attiya et al. [ABND95], its dynamic multi-writer counterparts by Lynch and Shvartsman [LS97, GLS10] and Englert and Shvartsman [ES00], wait-free simulations of Jayanti et al. [JCT98], low-latency atomic wait-free implementations of Dutta et al. [DGLV10] and Georgiou et al. [GNS09], and the consensus-free versions of Aguilera et al. [AKMS11]. These solutions are not directly applicable to our model where KVSs are used as base objects, due to the old-new overwrite problem.

Notable exceptions that are applicable in our KVS context are SWMR regular register emulation by Gafni and Lamport [GL03] and its Byzantine variant by Abraham et al. [ACKM06]

that use registers as base objects. However, transforming these SWMR emulations to support a large number of writers is inefficient: standard register transformations [AW04, CGR11] that can be used to this end require at least as many SWMR regular registers as there are clients, even if there are no faults. This is prohibitively expensive in terms of space complexity and effectively limits the number of supported clients. Chockler and Malkhi [CM05] acknowledge this issue and propose an algorithm that supports an unbounded number of clients (like our algorithm). However, their method uses base objects (called "active disks") that may carry out computations. In contrast, our emulation leverages the operations in the KVS interface, which is more general than a register due to its list and remove operations, and supports an unbounded number of clients. Ye et al. [YXYB10] overcome the GC racing problem by having the readers "reserve" the versions they intend to read, by storing extra values that signal to the garbage collector not to remove the version being read. This approach requires readers to have write access, which is not desirable.

Two recent works share our goal of providing robust storage from KVS base objects. Abu-Libdeh et al. [ALPW10] propose RACS, an approach that casts RAID techniques to the KVS context. RACS uses a model different from ours and basically relies on a proxy between the clients and the KVSs, which may become a bottleneck and single point-of-failure. In a variant that supports multiple proxies, the proxies communicate directly with each other for synchronizing their operations. Bessani et al. [BCQ+11] propose a distributed storage system, called DepSky, which employs erasure coding and cryptographic tools to store data on KVS objects prone to Byzantine faults. However, the basic version of DepSky allows only a single writer and thereby circumvents the problems addressed here. An extension supports multiple writers through a locking mechanism that determines a unique writer using communication among the clients. In comparison, the multi-writer versions of RACS and DepSky both serialize write operations, whereas our algorithm allows concurrent write operations from multiple clients in a wait-free manner. Therefore, our solution scales easily to a large number of clients.

## 5.6 Conclusion

The work described in this chapter offers a theoretical insight in how to build robust storage abstractions from unreliable key-value store (KVS) objects, as commonly provided by distributed cloud-storage systems over the Internet. These results complement the practical insights provided in the previous chapter.

In the first part of the chapter, we provided an emulation of a regular register over a set of atomic KVSs; it supports an unbounded number of clients that need not know each other and never interact directly.

The presented algorithm is wait-free and robust against the crash failure of a minority of the KVSs and of any number of clients. The algorithm stores versioned values under two types of keys — an eternal key that is never removed, and temporary keys that are dynamically added and removed. This novel mechanism allows garbage collection of obsolete values in parallel to wait-free client operations. Simulations and benchmarks with actual cloud-storage providers demonstrate that the algorithm works well under practical circumstances.

For ease of exposition, we have assumed atomic semantics of KVSs, but practical KVSs may only provide eventual consistency [Vog09]. In our practical experience, we never observed non-atomic behavior; note that some cloud providers already provide atomic operations [CWO+11].

In the second part of the chapter, we researched in detail the building block of a replicated storage — a technical foundation of KVS. We have shown that the consensus number of a typ-

ical storage replica in timestamp-based replication algorithms is infinite, but a KVS, provided by most cloud storage services, has consensus number one. Therefore these two providers have fundamentally different power for synchronizing operations of multiple clients in wait-free algorithms (formally captured in Theorem 1). This result explains why replication algorithms using KVS providers, such as DepSky [BCQ+11] and the algorithms from the beginning of the chapter, must use more complex methods to synchronize multiple clients than traditional data replication schemes.

Our result also gives an incentive for considering extensions of the KVS interface, such as the active KVS model introduced recently [GLK+10].

# Chapter 6

# State Machine Replication: The MOD-SMART BFT Replication Protocol

*Chapter Authors:*
João Sousa and Alysson Bessani (FFCUL).

## 6.1 Introduction

Replication is a fundamental technique for implementing dependable services that are able to ensure integrity and availability despite the occurrence of faults and intrusions. State Machine Replication (SMR) [Lam78,Sch90] is a popular replication method that enables a set of replicas (state machines) to execute the same sequence of operations for a service even if a fraction of the them are faulty.

A fundamental requirement of SMR is to make all client-issued requests to be totally ordered across replicas. Such requirement demands the implementation of a total order broadcast protocol, which is known to be equivalent the consensus problem [CNV06, HT94, MHS11]. Therefore, a solution to the consensus problem is in the core of any distributed SMR protocol.

In the last decade, many practical SMR protocols for the Byzantine fault model were published (e.g., [AEMGG+05,CL02,CML+06,KAD+07b,VCBL09,VCB+11]). However, despite their efficiency, such protocols are *monolithic*: they do not separate clearly the consensus primitive from the remaining protocol.

From a theoretical point of view, many Byzantine fault-tolerant (BFT) total order broadcast protocols (the main component of a BFT SMR implementation) were built using black-box Byzantine consensus primitives (e.g., [CKPS01,CNV06,HT94,MHS11]). This modularity simplifies the protocols, making them both easy to reason about and to implement. Unfortunately, these modular transformations plus the underlying consensus they use always require more communication steps than the aforementioned monolithic solutions.

Figure 6.1 presents the typical message pattern of modular BFT total order broadcast protocols when used to implement SMR. The key point of most of these transformations is the use of BFT reliable broadcast protocol [Bra84] to disseminate client requests among replicas, ensuring they will be eventually proposed (and decided) in some consensus instance that defines the order of messages to be executed. As illustrated in Figure 6.1, the usual BFT reliable broadcast requires three communication steps [Bra84].

It is known that optimally resilient Byzantine consensus protocols cannot safely decide a value in two or less communication steps [DGV05, MA05]. This means that latency-optimal protocols for BFT SMR that use only $3f + 1$ replicas to tolerate $f$ Byzantine faults (e.g., PBFT [CL02]) requires at least three communication steps for the consensus plus two extra

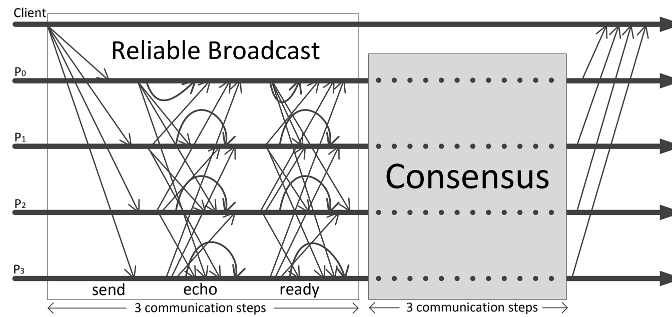



Figure 6.1: Modular BFT state machine replication message pattern for a protocol that uses reliable broadcast and a consensus primitives. This protocol is adapted from [MHS11], when tolerating a single fault.

steps to receive the request from the client and send a reply[1]. By the other hand, the protocol of Figure 6.1 requires at least six communication steps to totally order a message in the best-case, plus one more to send a reply to the client, making a total of seven steps.

Considering this gap, in this chapter we investigate the following question: *Is it possible to obtain a BFT state machine replication protocol with an optimal number of communications steps (similar to PBFT), while explicitly using a consensus primitive at its core?* The main contribution of this work is a new transformation from Byzantine consensus to BFT state machine replication dubbed *Modular State Machine Replication* (MOD-SMART), which answers this question affirmatively. MOD-SMART implements SMR using a special Byzantine consensus primitive called *Validated and Provable Consensus* (VP-Consensus), which can be easily obtained by modifying existing leader-driven consensus algorithms (e.g., [Cac09, Lam01, MA05, RMS10, Zie04]). To our knowledge, MOD-SMART is the first modular BFT SMR protocol built over a well-defined consensus module which requires only the optimal number of communication steps, i.e., the number of communication steps of consensus plus two.

The core of our solution is the definition and use of the VP-Consensus as a *"grey-box" abstraction* that allows the modular implementation of SMR without using reliable broadcast, thus avoiding the extra communication steps required to safely guarantee that all requests arrive at all correct replicas. The monolithic protocols, on the other hand, avoid those extra steps by merging the reliable broadcast with the consensus protocol, being thus more complex. MOD-SMART avoids mixing protocols by using the rich interface exported by VP-Consensus, that allows it to handle request timeouts and, if needed, triggers internal consensus timeouts. The use of a VP-Consensus is a good compromise between modularity and efficiency, specially because this primitive can be easily implemented with simple modifications on several leader-driven partially-synchronous Byzantine consensus protocols [Cac09, Lam01, LM07, MA05, RMS10, Zie04].

Although this work main contribution is theoretical, our motivation is very practical and highly relevant to the TClouds project. MOD-SMART is implemented as one of the core modules of BFT-SMART [LaS10] (described in Chapter 8 of D2.2.1), an open-source Java-based BFT SMR library in which modularity is treated as a first-class property.

The chapter is organized in the following way. We first describe our system model and the

[1]This excludes optimistic protocols that are very efficient in contention-free executions [AEMGG+05, CML+06], speculative protocols [KAD+07b], protocols that rely on trusted components [VCB+11], and fast protocols that require more than $3f + 1$ replicas [MA05].

problem we want to address in Sections 6.2 and 6.3. The Validated and Provable Consensus primitive is discussed in Section 6.4. Next, Section 6.5 present the the MOD-SMART algorithms. Possible optimizations and additional considerations are discussed in Section 6.6. In Sections 6.7 and 6.8 we put the related work in context and present our conclusions. Finally, all proofs that MOD-SMART implements SMR are described in Appendix C.

## 6.2 System Model

We consider a system composed by a set of $n \geq 3f + 1$ replicas $R$, where a maximum of $f$ replicas may be subject to *Byzantine faults*, and a set $C$ with an unbounded (but finite) number of clients, which can also suffer Byzantine faults. A process (client or replica) is considered correct if it never deviates from its specification; otherwise, it is considered faulty.

Like in PBFT and similar protocols [CL02, CML+06, KAD+07b, VCBL09], MOD-SMART does not require synchrony to assure *safety*. However, it requires synchrony to provide *liveness*. This means that, even in the presence of faults, correct replicas will never evolve into an inconsistent state; but the execution of the protocol is guaranteed to terminate only when the system becomes synchronous. Due to this, we assume an *eventually synchronous* system model [DLS88]. In such model, the system operates asynchronously until some unknown instant, at which it will become synchronous. At this point, unknown time bounds for computation and communication will be respected by the system.

We further assume that all processes communicate through *reliable and authenticated point-to-point channels*, that can be easily implemented over fair links using retransmission and message authentication codes.

Finally, we assume the existence of cryptographic functions that provide digital signatures, message digests, and message authentication codes (MAC).

## 6.3 State Machine Replication

The state machine replication model was first proposed in [Lam78], and later generalized in [Sch90]. In this model, an arbitrary number of client processes issue commands to a set of replica processes. These replicas implement a stateful service that changes its state after processing client commands, and sends replies to the clients that issued them. The goal of this technique is to make the state at each replica evolve in a consistent way, thus making the service completely and accurately replicated at each replica. In order to achieve this behavior, it is necessary to satisfy four properties:

1. If any two correct replicas $r$ and $r'$ apply operation $o$ to state $s$, both $r$ and $r'$ will reach state $s'$;
2. Any two correct replicas $r$ and $r'$ start with state $s_0$;
3. Any two correct replicas $r$ and $r'$ execute the same sequence of operations $o_0, ..., o_i$;
4. Operations from correct clients are always executed.

The first two requirements can be fulfilled without any distributed protocol, but the following two directly translates to the implementation of a total order broadcast protocol – which

is equivalent to solving the consensus problem. MOD-SMART satisfy properties 3 and 4, assuming the existence of a VP-Consensus primitive and that the service being replicated respects properties 1 and 2.

## 6.4 Validated and Provable Consensus

In this section we introduce the concept of *Validated and Provable Consensus* (VP-Consensus). By 'Validated', we mean the protocol receives a predicate $\gamma$ together with the proposed value – which any decided value must satisfy. By 'Provable', we mean that the protocol generates a cryptographic proof $\Gamma$ that certifies that a value $v$ was decided in a consensus instance $i$. More precisely, a VP-Consensus implementation offers the following interface:

- *VP-Propose(*$i, l, \gamma, v$*)*: proposes a value $v$ in consensus instance $i$, with initial leader $l$ and predicate $\gamma$;
- *VP-Decide(*$i, v, \Gamma$*)*: triggered when value $v$ with proof $\Gamma$ is decided in consensus instance $i$;
- *VP-Timeout(*$i, l$*)*: used to trigger a timeout in the consensus instance $i$, and appoint a new leader process $l$.

Three important things should be noted about this interface. First, VP-Consensus assumes a leader-driven protocol, similar to any Byzantine Paxos consensus. Second, the interface assumes the VP-Consensus implementation can handle timeouts to change leaders, and a new leader is (locally) chosen after a timeout. Finally, we implicitly assume that all correct processes will invoke *VP-Propose* for an instance $i$ using the same predicate $\gamma$.

Just like usual definitions of consensus [Cac09, CNV06, HT94], VP-Consensus respects the following properties:

- *Termination*: Every correct process eventually decides;
- *Integrity*: No correct process decides twice;
- *Agreement*: No two correct processes decide differently.

Moreover, two additional properties are also required:

- *External Validity*: If a correct process decides $v$, then $\gamma(v)$ is true;
- *External Provability*: If some correct process decides $v$ with proof $\Gamma$ in a consensus instance $i$, all correct process can verify that $v$ is the decision of $i$ using $\Gamma$.

*External Validity* was originally proposed by Cachin et al. [CKPS01], but we use a slightly modified definition. In particular, *External Validity* no longer explicitly demands validation data for proposing $v$, because such data is already included in the proposed value, as will be clear in Section 6.5.

### 6.4.1 Implementation requirements

Even though our primitive offers the classical properties of consensus, the interface imposes some changes in its implementation. Notice that we are not trying to specify a new consensus algorithm; we are only specifying a primitive that can be obtained by making simple modifications to existing ones [Cac09,Lam01,LM07,MA05,RMS10]. However, as described before, our interface assumes that such algorithms are leader-driven and assume the partially synchronous system model. Most Paxos-based protocols satisfy these conditions [Cac09, MA05, RMS10, Zie04], and thus can be used with MOD-SMART. In this section we present an overview of the required modifications on consensus protocols, without providing explanations for it. We will come back to the modifications in Section 6.5.5, when it will become clear why they are required.

The first change is related to the timers needed in the presence of partial synchrony. To our knowledge, all published algorithms for such system model requires a timer to ensure liveness despite leader failures [Cac09, Lam98, MA05]. The primitive still needs such timer; but it will not be its responsibility to manage it. Instead, we invoke *VP-Timeout* to indicate to the consensus that a timeout has occurred, and it needs to handle it.

The second change is related to the assumption of a leader-driven consensus. To our knowledge, all the leader-driven algorithms in literature have deterministic mechanisms to select a new leader when sufficiently many of them suspect the current one. These suspicions are triggered by a timeout. A VP-Consensus implementation still requires the election of a new leader upon a timeout. However, the next leader will be defined by MOD-SMART, and is passed as an argument in the *VP-Propose* and *VP-Timeout* calls. Notice that these two requirements are equivalent to assuming the consensus protocol requires a leader election module, just like $\Omega$ failure detector, which is already used in some algorithms [MA05,Cac09].

The third change imposes the consensus algorithm to generate the cryptographic proof $\Gamma$ to fulfill the *External Provability* property. This proof can be generated by signing the messages that can trigger a decision of the consensus[2]. An example of proofs would be a set of $2f + 1$ signed COMMIT messages in PBFT [CL02] or $\lceil (n + f + 1)/2 \rceil$ signed COMMITPROOF messages in Parametrized FaB [MA05].

Finally, we require each correct process running the consensus algorithm to verify if the value being proposed by the leader satisfies $\gamma$ before it is accepted. Correct processes must only accept values that satisfy such predicate and discard others – thus fulfilling the *External Validity* property.

## 6.5 The MOD-SMART Algorithm

In this section we describe MOD-SMART, our modular BFT state machine replication algorithm. The protocol is divided into three sub-algorithms: client operation, normal phase, and synchronization phase. The proofs that MOD-SMART satisfies the BFT state machine replication properties under our system model are presented in the Appendix.

[2] Due to the cost of producing digital signatures, the cryptographic proof can be generated with MAC vectors instead of digital signatures, just like in PBFT [CL02].

### 6.5.1 Overview

The general architecture of a replica is described in Figure 6.2. MOD-SMART is built on top of a reliable and authenticated point-to-point communication substrate and a VP-Consensus implementation. Such module may also use the same communication support to exchange messages among processes. MOD-SMART uses VP-Consensus to execute a sequence of consensus instances, where in each instance $i$ a batch of operations are proposed for execution, and the same proposed batch is decided on each correct replica. This is the mechanism by which we are able to achieve total order across correct replicas.

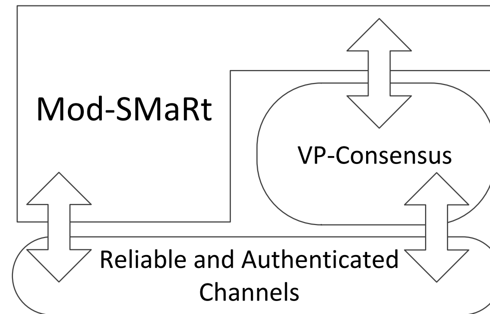



Figure 6.2: MOD-SMART replica architecture. The reliable and authenticated channels layer guarantee the delivery of point-to-point messages, while the VP-Consensus module is used to establish agreement on the message(s) to be delivered in an consensus instance.

During normal phase, a log of the decided values is constructed based on the sequence of VP-Consensus executions. Each log entry contains the decided value, the *id* of the consensus instance where it was decided, and its associated proof. To simplify our design, MOD-SMART assumes each correct replica can execute concurrently only the current instance $i$ and previous consensus instance $i-1$. All correct replicas remain available to participate in consensus instance $i-1$, even if they are already executing $i$. This is required to ensure that if there is one correct replica running consensus $i-1$ but not $i$, there will be at least $n-f$ correct replicas executing $i-1$, which ensures the delayed replica will be able to finish $i-1$.

Due to the asynchrony of the system, it is possible that a replica receives messages for a consensus instance $j$ such that $j > i$ (early message) or $j < i-1$ (outdated message). Early messages are stored in an out-of-context buffer for future processing while outdated messages are discarded. We do not provide pseudo-code for this mechanism, relying on our communication layer to deliver messages in accordance with the consensus instances being executed.

This pretty much describes the *normal phase* of the protocol, which is executed in the absence of faults and in the presence of synchrony. When these conditions are not satisfied, the *synchronization phase* might be triggered.

MOD-SMART makes use of the concept of *regencies*. This is equivalent to the *view* mechanism employed by PBFT and ViewStamped Replication [CL02, OL88], where a single replica will be assigned as the leader for each regency. Such leader will be needed both in MOD-SMART, and in the VP-Consensus module. During each regency, the normal case operation can be repeated infinitely; during a synchronization phase, an unbounded (but finite) number of regency changes can take place, since the system will eventually become synchronous.

The avoidance of executing a reliable multicast before starting the Byzantine consensus may lead to two problems. First, a faulty leader may not propose messages from some client

for ordering, making it starve. Second, a faulty client can send messages to all replicas but to the current (correct) leader, making other replicas suspect it for not ordering messages from this client. The solution for these problems is to suspect the leader only if the timer associated with a message expires twice, making processes forward the pending message to the leader upon the first expiration.

In case a regency change is needed (i.e., the leader is suspected), timeouts will be triggered at all replicas and the synchronization phase will take place. During this phase, MOD-SMART must ensure three properties: (1) a quorum of $n - f$ replicas must have the pending messages that caused the timeouts; (2) correct replicas must exchange logs to jump to the same consensus instance; and (3) a timeout is triggered in this consensus, proposing the same leader at all correct replicas (the one chosen during the regency change). Notice that MOD-SMART does not verify consensus values to ensure consistency: all these checks are done inside of the VP-Consensus module, after its timeout is triggered. This substantially simplifies faulty leader recovery by breaking the problem in two self-contained blocks: the state machine replication layer ensures all processes are executing the same consensus with the same leader while VP-Consensus deals with the inconsistencies within a consensus.

### 6.5.2 Client Operation

Algorithm 10 describes how the client invokes an operation in MOD-SMART. When a client wants to issue a request to the replicas, it sends a REQUEST message in the format specified (line 6). This message contains the sequence number for the request and the command issued by the client. The inclusion of a sequence number is meant to uniquely identify the command (together with the client id), and prevent replay attacks made by an adversary that might be sniffing the communication channel. A digital signature $\alpha_c$ is appended to the message to prove that such message was produced by client $c$. Although this signature is not required, its use makes the system resilient against certain attacks [ACKL08, CWA+09].

The client waits for at least $f + 1$ matching replies from different replicas, for the same sequence number (lines 9–11), and return the operation result.

**Algorithm 10:** Client-side protocol for client $c$.

```
1  Upon Init do
2      nextSeq = 0
3      Replies ← ∅

4  Upon Invoke(op) do
5      nextSeq = nextSeq + 1
6      send ⟨REQUEST, nextSeq, op⟩_αc to R

7  Upon reception of ⟨REPLY, seq, rep⟩ from r ∈ R do
8      Replies ← Replies ∪ {⟨r, seq, rep⟩}
9      if ∃ seq, rep : |{⟨∗, seq, rep⟩ ∈ Replies}| > f
10         Replies ← Replies\{⟨∗, seq, rep⟩}
11         return rep
```

### 6.5.3 Normal Phase

The normal phase is described in Algorithm 11, and its message pattern is illustrated in Figure 6.3. The goal of this phase is to execute a sequence of consensus instances in each replica. The values proposed by each replica will be a batch of operations issued by the clients. Because

each correct replica executes the same sequence of consensus instances, the values decided in each instance will be the same in all correct replicas, and since they are batches of operations, they will be *totally ordered* across correct replicas. All variables and functions used by the replicas in Algorithms 11 and 12 are described in Table 6.1.

Reception of client requests are processed in line 1-2 through procedure $RequestReceived$ (lines 20–24). Requests are only considered by correct replicas if the message contains a valid signature and the sequence number expected from this client (to avoid replay attacks), as checked in line 21. If a replica accepts an operation issued by a client, it stores it in the $ToOrder$ set, activating a timer associated with the request (lines 22–24). Notice that a message is also accepted if it is forwarded by other replicas (lines 18-19).

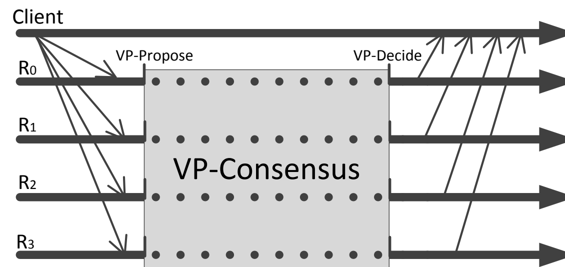



Figure 6.3: Communication pattern of MOD-SMART normal phase for $f = 1$. A correct client send an operation to all replicas, a consensus instance is executed to establish total order, the operation is executed, and a reply is sent to the client.

When the $ToOrder$ set contains some request to be ordered, there is no consensus being executed and the ordering of messages is not stopped (see next section), a sub-set of operations $Batch$ from $ToOrder$ is selected to be ordered (lines 3 and 4). The predicate *fair* ensures that all clients with pending requests will have approximately the same number of operations in a batch to avoid starvation. The replica will then create a consensus instance, using $Batch$ as the proposed value (lines 5 and 6). The predicate $\gamma$ given as an argument in *VP-Propose* should return TRUE for a proposed value $V$ if the following three conditions are met:

1. $fair(V)$ is TRUE (thus $V$ is not an empty set);
2. Each message in $V$ is either in the $ToOrder$ set of the replica or is correctly signed and contains the next sequence number expected from the client that issued the operation;
3. Each message in $V$ contains a valid command with respect to the service implemented by MOD-SMART.

When a consensus instance decides a value (i.e., a batch of operations) and produces its corresponding proof (line 7), MOD-SMART will: store the batch of operations and its cryptographic proof in each replica log (line 11); cancel the timers associated with each decided request (line 14); deterministically deliver each operation contained in the batch to the application (line 16); and send a reply to the client that requested the operation with the corresponding response (line 17). Notice that if the algorithm is stopped (possibly because the replica is running a synchronization phase, see next section), decided messages are stored in a $Decided$ set (lines 8 and 9), instead of being executed.

Table 6.1: Variables and functions used in Algorithms 11 and 12.

| *Variables* | | |
|---|---|---|
| Name | Initial Value | Description |
| $timeout$ | INITIAL_TIMEOUT | Timeout for a message to be ordered. |
| $maxBatch$ | MAX_BATCH | Maximum number of operations that a batch may contain. |
| $creg$ | 0 | Replica current regency. |
| $nreg$ | 0 | Replica next regency. |
| $currentCons$ | -1 | Current consensus being executed. |
| $DecLog$ | $\emptyset$ | Log of all decided consensus instances and their proofs. |
| $ToOrder$ | $\emptyset$ | Pending messages to be ordered. |
| $Tmp$ | $\emptyset$ | Messages collected in a STOP messages. |
| $Decided$ | $\emptyset$ | Decision values obtained during the synchronization phase. |
| $stopped$ | FALSE | Indicates if the synchronization phase is activated. |
| $lastSeq[1..\infty]$ | $\forall c \in C : lastSeq[c] \leftarrow 0$ | Last request sequence number used by each client $c$. |
| $ChangeReg[1..\infty]$ | $\forall g \in N : ChangeReg[g] \leftarrow \emptyset$ | Replicas that want a change to regency $g$. |
| $Data[1..\infty]$ | $\forall g \in N : Data[g] \leftarrow \emptyset$ | Signed STOPDATA messages collected by the leader during change to regency $g$. |
| $Sync[1..\infty]$ | $\forall g \in N : Sync[g] \leftarrow \emptyset$ | Set of Logs sent by the leader to all replicas during regency change $g$. |

| *Functions* | |
|---|---|
| Interface | Description |
| `activateTimers(Reqs, timeout)` | Creates a timer for each request in *Reqs* with value *timeout*. |
| `cancelTimers(Reqs)` | Cancels the timer associated with each request in *Reqs*. |
| `execute(op)` | Makes the application execute operation *op*, returning the result. |
| `validSig(req)` | Returns TRUE if request *eq* is correctly signed. |
| `noGaps(Log)` | Returns TRUE if sequence of consensus *Log* does not contain any gaps. |
| `validDec(decision)` | Returns TRUE if *decision* contains a valid proof. |
| `hCons(Log)` | Returns the consensus instance from *Log* with highest id. |
| `hLog(Logs)` | Returns the largest log contained in *Logs*. |

**Algorithm 11:** Normal phase at replica $r$.

1 **Upon** reception of $m = \langle \text{REQUEST}, seq, op\rangle\alpha_c$ from $c \in C$ **do**
2   `RequestReceived` $(m)$

3 **Upon** $(toOrder \neq \emptyset) \wedge (currentCons = -1) \wedge (\neg stopped)$ **do**
4   $Batch \leftarrow X \subseteq ToOrder : |X| \leq maxBatch \wedge \texttt{fair}(X)$
5   $currentCons \leftarrow \texttt{hCons}(DecLog).i + 1$
6   `VP-Propose` $(currentCons, creg \bmod R, \gamma, Batch)$

7 **Upon** *VP-Decide*$\langle i, Batch, Proof\rangle$ **do**
8   **if** $stopped$
9     $Decided \leftarrow Decided \cup \{\langle i, Batch, Proof\rangle\}$
10   **else**
11     $DecLog \leftarrow DecLog \cup \{\langle i, Batch, Proof\rangle\}$
12     **if** $currentCons = i$ **then** $currentCons \leftarrow -1$
     // Deterministic cycle
13     **foreach** $m = \langle \text{REQUEST}, seq, op\rangle_{\alpha_c} \in Batch$ **do**
14       `cancelTimers` $(\{m\})$
15       $ToOrder \leftarrow ToOrder \setminus \{m\}$
16       $rep \leftarrow$ `execute` $(op)$
17       send $\langle \text{REPLY}, seq, rep\rangle$ to $c$

18 **Upon** reception of $\langle \text{FORWARDED}, M\rangle$ from $r' \in R$ **do**
19   $\forall m \in M :$ `RequestReceived` $(m)$

20 **Procedure** *RequestReceived(m)*
21   **if** $lastSeq[c] + 1 = m.seq \wedge \texttt{validSig}(m)$
22     $ToOrder \leftarrow ToOrder \cup \{m\}$
23     **if** $\neg stopped$ **then** $\texttt{activateTimers}(\{m\}, timeout)$
24     $lastSeq[c] \leftarrow m.seq$

### 6.5.4 Synchronization Phase

The synchronization phase is described in Algorithm 12, and its message pattern is illustrated in Figure 6.4. This phase aims to perform a regency change and force correct replicas to synchro-

nize their states and go to the same consensus instance. It occurs when the system is passing through a period of asynchrony, or there is a faulty leader that does not deliver client requests before their associated timers expire. This phase is started when a *timeout* event is triggered for a sub-set $M$ of pending messages in $ToOrder$ (line 1).

When the timers associated with a set of requests $M$ are triggered for the first time, the requests are forwarded to all replicas (lines 2 and 3). This is done because a faulty client may have sent its operation only to some of the replicas, therefore starting a consensus in less than $n - f$ of them (which is not sufficient to ensure progress, and therefore will cause a timeout in these replicas). This step forces such requests to reach all correct replicas, without forcing a leader change.

If there is a second timeout for the same request, the replica starts a regency change (line 4). When a regency change begins in a replica, the processing of decisions is stopped (line 7), the timers for all pending requests are canceled (line 9) and a $\mathrm{STOP}$ message is sent to all replicas (line 10). This message informs other replicas that a timeout for a given set of requests has occurred. When a replica receives more than $f$ $\mathrm{STOP}$ messages requesting the next regency to be started (line 15), it begins to change its current regency using the valid messages in $Tmp$ (line 16). This procedure ensures that a correct replica starts a view change as soon as it knows that at least one correct replica started it, even if no timeout was triggered locally.

When a replica receives more than $2f$ $\mathrm{STOP}$ messages, it will install the next regency (lines 19 and 20). It is necessary to wait at least $2f + 1$ messages to make sure that eventually all correct replicas will install the next regency. Following this, the timers for all operations in the $ToOrder$ set will be re-activated and a new leader will be elected (lines 21–23).

After the next regency is installed, it is necessary to force all replicas to go to the same state (i.e., synchronize their logs and execute the logged requests) and, if necessary, start the consensus instance. To accomplish this, all replicas send a $\mathrm{STOPDATA}$ message to the new regency leader, providing it with their decision log (line 23). As long as the proof associated with each decided value is valid and there is no consensus instance missing, the leader will collect these messages (lines 26 and 27). This is necessary because it proves that each consensus instances has decided some batch of operations (which will be important later). When at least $n - f$ valid $\mathrm{STOPDATA}$ messages are received by the leader, it will send a $\mathrm{SYNC}$ message to all replicas, containing all the information gathered about their decided instances in at least $n - f$ replicas (lines 28 and 29).

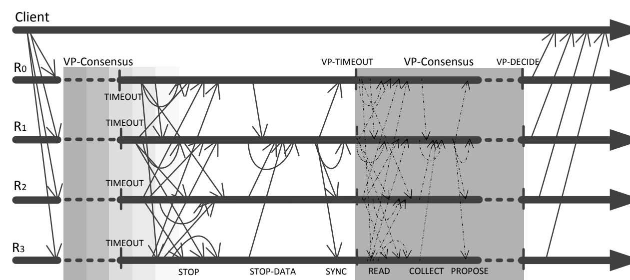



Figure 6.4: Communication steps of synchronization phase for $f = 1$. This phase is started when the timeout for a message is triggered for a second time, and can run simultaneously with VP-Consensus. Dashed arrows correspond to messages of the VP-Consensus protocol.

When a replica receives a $\mathrm{SYNC}$ message, it executes the same computations performed by the leader (lines 31–35) to verify if the leader has gathered and sent valid information. If the

leader is correct, after receiving the same SYNC message, all correct replicas will choose the same highest log (line 36) and resume decision processing (line 37). All correct replicas will evolve into the same state as they deliver the value of each consensus instance that was already decided in other replicas (lines 40 and 41) and either trigger a timeout in the VP-Consensus being executed (line 42 and 43) or make everything ready to start a new consensus instance (line 44).

### 6.5.5 Reasoning about the Consensus Modifications

As we mentioned in Section 6.4.1, the VP-Consensus primitive does not need to start and stop timers, since our state machine algorithm already does that. Due to this, the VP-Consensus module only needs to be notified by the state machine algorithm when it needs to handle a timeout. This is done by invoking *VP-Timeout* for a consensus $i$, at the end of a synchronization phase (line 43 of Algorithm 12). The *VP-Timeout* operation also receives as an argument the new leader the replica should rely on. This is needed because we assume a leader-driven consensus, and such algorithms tend to elect the leader in a coordinated manner. But when a delayed replica jumps from an old consensus to a consensus $i$ during the synchronization phase, it will be out-of-sync with respect to the current regency, when compared with the majority of replicas that have already started consensus $i$ during the normal phase. For this reason, we need to explicitly inform VP-Consensus about the new leader.

Let us now discuss why the *External Validity* is required for MOD-SMART. The classic *Validity* property would be sufficient in the crash fault model, because processes are assumed to fail only by stopping, and will not propose invalid values; however, in the Byzantine fault model such behavior is permitted. A faulty process may propose an invalid value, and such value might be decided. An example of such value can be an empty batch. This is a case that can prevent progress within the algorithm. By forcing the consensus primitive to decide a value that is useful for the algorithm to keep making progress, we can prevent such scenario from occurring, and guarantee liveness as long as the execution is synchronous.

Finally, it should now be clear why the *External Provability* property is necessary: in the Byzantine fault model, replicas can lie about which consensus instance they have actually finished executing, and also provide a fake/corrupted decision value if a synchronization phase is triggered. By forcing the consensus primitive to provide a proof, we can prevent faulty replicas from lying. The worst thing a faulty replica can do is to send old proofs from previous consensus. However, since MOD-SMART requires at least $n - f$ logs from different replicas, there will be always more than $f$ up-to-date correct replicas that will provide their most recent consensus decision.

## 6.6 Optimizations

In this section we discuss a set of optimizations for efficient MOD-SMART implementation. The first important optimization is related with bounding the size of the decision log. In MOD-SMART, such log can grow indefinitely, making it inappropriate for real systems. To avoid this behavior we propose the use of checkpoints and state transfer. Checkpoints would be performed periodically in each replica: after some number *D* of decisions are delivered, the replica request the state from the application, save it in memory or disk, and clear the log up to this point[3]. If

[3] Notice that, differently from the PBFT checkpoint protocol [CL02], MOD-SMART checkpoints are local operations.

**Algorithm 12:** Synchronization phase at replica $r$.

1 **Upon** timeout for requests $M$ **do**
2   $M_{\text{first}} \leftarrow \{m \in M : \text{first timeout of } m\}$
3   **if** $M_{\text{first}} \neq \emptyset$ **then** `send` $\langle \text{FORWARDED}, M \rangle$ to R
4   **else if** $M \backslash M_{\text{first}} \neq \emptyset$ **then** `StartRegChange` $(M \backslash M_{\text{first}})$

5 **Procedure** *StartRegChange($M$)*
6   **if** $nreg = creg$
7     $stopped \leftarrow$ TRUE
8     $nreg \leftarrow creg + 1$
9     `cancelTimers`$(ToOrder)$ // Cancel all timers
10     `send` $\langle \text{STOP}, nreg, M \rangle$ to $R$

11 **Upon** reception of $\langle \text{STOP}, reg, M \rangle$ from $r' \in R$ **do**
12   **if** $reg = creg + 1$
13     $Tmp \leftarrow Tmp \cup M$
14     $ChangeReg[reg] \leftarrow ChangeReg[reg] \cup \{r'\}$
15     **if** $|ChangeReg[reg]| > f$
16       $M' \leftarrow \{m \in Tmp : m.seq > lastSeq[m.c] \wedge$ `validSig`$(m)\}$
17       `StartRegChange` $(M')$
18       $ToOrder \leftarrow ToOrder \cup M'$
19       **if** $|ChangeReg[reg]| > 2f \wedge nreg > creg$
20         $creg \leftarrow nreg$
21         `activateTimers` $(ToOrder, timeout)$
22         $leader \leftarrow creg \bmod n$
23         `send` $\langle \text{STOPDATA}, reg, DecLog \rangle_{\alpha_r}$ to $leader$

24 **Upon** recept. of $m = \langle \text{STOPDATA}, creg, Log \rangle_{\alpha_{r'}}$ from $r' \in R$ **do**
25   **if** $creg \bmod n = r$
26     **if** $($`noGaps`$(Log)) \wedge (\forall d \in Log :$ `validDec`$(d))$
27       $Data[creg] \leftarrow Data[creg] \cup \{m\}$
28     **if** $|Data[creg]| \geq n - f$
29       `send` $\langle \text{SYNC}, creg, Data[creg] \rangle$ to $R$

30 **Upon** reception of $\langle \text{SYNC}, creg, Proofs \rangle$ from $r' \in R$ **do**
31   **if** $(nreg = creg) \wedge (creg \bmod n = r') \wedge ProofCons[creg] = \emptyset$
32     **foreach** $\langle \text{STOPDATA}, creg, Log \rangle_{\alpha_{r''}} \in Proofs$ **do**
33       **if** $($`noGaps`$(Log)) \wedge (\forall d \in Log :$ `validDec`$(d))$
34         $Sync[creg] \leftarrow Sync[creg] \cup \{\langle r'', Log \rangle\}$
35     **if** $|Sync[creg]| \geq n - f$
36       $Log \leftarrow$ `hLog`$(Sync[creg] \cup \{\langle r, DecLog \rangle\}) \cup Decided$
37       $stopped \leftarrow$ FALSE
38       $Decided \leftarrow \emptyset$
39       $Tmp \leftarrow \emptyset$
      // Deterministic cycle
40       **foreach** $\langle i', B, P \rangle \in Log : i' >$ `hCons`$(DecLog).i$ **do**
41         `Trigger` VP-Decide$\langle i', B, P \rangle$
42       **if** $currentCons =$ `hCons`$(Log).i + 1$
43         `VP-Timeout` $(currentCons, creg \bmod R)$
44       **else** $currentCons = -1$

in the end of a synchronization phase a replica detects a gap between the latest decision of its own log, and the latest decision of the log it chose, it invokes a state transfer protocol. Such a protocol would request from the other replicas the state that was saved in their latest checkpoint. Upon the reception of $f+1$ matching states from different replicas, the protocol would force the application to install the new state, and resume execution.

The second optimization aims to avoid the computational cost of generating and verifying digital signatures in the protocol critical path: client requests and VP-Consensus proofs (to satisfy External Provability) can be signed using MAC vectors instead of digital signatures, as done in PBFT. However, in the case of client requests, this results in a less robust state machine implementation vulnerable to certain performance degradation attacks [ACKL08, CWA+09].

If we use VP-Consensus based on a Byzantine consensus algorithm matching the generalization given in [Lam01], and employ the optimizations just described, MOD-SMART matches the message pattern of PBFT in synchronous executions with correct leaders, requiring thus same number of communication steps and cryptographic operations. This is exactly what was done in BFT-SMART [LaS10], an implementation of optimized MOD-SMART using the Byzantine consensus protocol described in [Cac09].

## 6.7 Related Work

Byzantine Fault Tolerance has gained wide-spread interest among the research community ever since Castro and Liskov showed that state machine replication can be practically accomplished for such fault model [CL02]. Their algorithm, best known as PBFT (Practical Byzantine Fault Tolerance) requires $3f+1$ replicas to tolerate $f$ Byzantine faults and is live under the partial synchronous system model [DLS88] (no synchrony is needed for safety). PBFT is considered the baseline for all BFT protocols published afterwards.

One of the protocols published following PBFT was Query/Update (Q/U) [AEMGG+05], an optimistic quorum-based protocol that presents better throughput with larger number of replicas than other agreement-based protocols. However, given its optimistic nature, Q/U performs poorly under contention, and requires $5f+1$ replicas. To overcome these drawbacks, Cowling et al. proposed HQ [CML+06], a hybrid Byzantine fault-tolerant SMR protocol similar to Q/U in the absence of contention. However, unlike Q/U, HQ only requires $3f+1$ replicas and relies on PBFT to resolve conflicts when contention among clients is detected. Following Q/U and HQ, Kotla et al. proposed Zyzzyva [KAD+07b], a speculative Byzantine fault tolerant protocol, which is considered to be one of the fastest BFT protocol up to date. It is worth noticing that all these protocols tend to be more efficient than PBFT because they avoid the complete execution of a consensus protocol in the expected normal case, relying on it only to solve exceptional cases.

Guerraoui et al. [GKQV10] proposed a well-defined modular abstraction unifying the optimizations proposed by previous protocols through composition, making it easy to design new protocols that are optimal in well-behaved executions (e.g., synchrony, absence of contention, no faults), but revert to PBFT if such nice behavior does not hold. However, the modularity proposed is at state machine replication level, in the sense that each module provides a way to totally order client requests under certain conditions, and does not suggest any clear separation between total order broadcast and consensus.

The relationship between total order broadcast and consensus for the Byzantine fault model is studied in many papers. Cachin et al. [CKPS01] show how to obtain total order broadcast from consensus provided that the latter satisfy the *External Validity* property, as needed with

MOD-SMART. Their transformation requires an echo broadcast plus public-key signature, adding thus at least two communication steps (plus the cryptography delay) to the consensus protocol. Correia et al. [CNV06] proposed a similar reduction without relying on public-key signatures, but using a reliable broadcast and a multi-valued consensus that satisfies a validity property different from Cachin's. The resulting transformation adds at least three communication steps to the consensus protocol in the best case. In a very recent paper, Milosevic et al. [MHS11] take in consideration many variants of the Byzantine consensus *Validity* property proposed in the literature, and show which of them are sufficient to implement total order broadcast. They also prove that if a consensus primitive satisfy the *Validity* property proposed in [DH08], then it is possible to obtain a reduction of total order broadcast to consensus with constant time complexity – which is not the case of the previous reductions in [CKPS01, CNV06]. However, their transformation still requires a reliable broadcast, and thus adds at least three communication steps to the consensus protocol. Doudou et al. [DGG05] show how to implement BFT total order broadcast with a weak interactive consistency (WIC) primitive, in which the decision comprises a vector of proposed values, in a similar way to a vector consensus (see, e.g., [CNV06]). They argue that the WIC primitive offers better guarantees than a Byzantine consensus primitive, eliminating the issue of the *Validity* property of consensus. The overhead of this transformation is similar to [CKPS01]: echo broadcast plus public-key signature.

All these works provide reductions from total order broadcast to Byzantine consensus by constructing a protocol stack that does not take into account the implementation of the consensus primitive; they only specify which properties such primitive should offer — in particular, they require some strong variant of the *Validity* property. MOD-SMART requires both a specific kind of *Validity* property, as well as a richer interface, as defined by our VP-Consensus abstraction. The result is a transformation that adds at most one communication step to implement total order broadcast, thus matching the number of communication steps of PBFT at the cost of using such gray-box consensus abstraction.

There are many works dedicated to generalize the algorithms of consensus. Lampson proposed an abstract Paxos algorithm, from which several other versions of Paxos can be derived (e.g., Byzantine, classic, and disk paxos) [Lam01]. Another generalization of Paxos-style protocols is presented in [LM07], where the protocol is reduced to a write-once register satisfying a special set properties. Implementations of such register are given for different system and failures models. Rütti et al. extends these works in [RMS10], where they propose a more generic construction than in [Lam01], and identify three classes of consensus algorithms. Finally, Cachin proposes a simple and elegant modular decomposition of Paxos-like protocols [Cac09] and shows how to obtain implementations of consensus tolerating crash or Byzantine faults based in the factored modules. All these works aim to modularize Paxos either for implementing consensus [Cac09, LM07, RMS10] or state machine replication [Lam01] under different assumptions; our work, on the other hand, aims at using a special kind of consensus to obtain a BFT state machine replication.

## 6.8 Conclusion

Despite the existence of several works providing efficient BFT state machine replication, none of them encapsulate the agreement within a consensus primitive, being thus monolithic. On the other hand, all published modular protocol stacks implementing BFT total order broadcast from Byzantine consensus require a number of communication steps greater than all practical BFT SMR. We bridge this gap by presenting MOD-SMART, a latency- and resiliency-optimal BFT

state machine replication algorithm that achieves modularity using a well-defined consensus primitive. To achieve such optimality, we introduce the *Validated and Provable Consensus* abstraction, which can be implemented by making simple modifications on existing consensus protocols. The protocol here presented is currently in use in BFT-SMART, an open-source BFT SMR library [LaS10].

# Chapter 7

# Extensible ZooKeeper

*Chapter Authors:*
Rüdiger Kapitza (TUBS) and Tobias Distler (FAU).

## 7.1 Introduction

Large-scale applications running on today's cloud infrastructures may comprise a multitude of processes distributed over a large number of nodes. Given these circumstances, fault-tolerant coordination of processes, although being an essential factor for the correctness of an application, is difficult to achieve. As a result, and to facilitate their design, fewer and fewer of such applications implement coordination primitives themselves, instead they rely on external coordination services. Large-scale distributed storage systems like BigTable [CDG+06] and HBase [Apa], for example, do not provide means for leader election but perform this task using the functionality of Chubby [Bur06] and ZooKeeper [HKJR10], respectively.

However, instead of implementing more complex services (e. g., leader election) directly, state-of-the-art coordination middleware systems only provides a basic set of low-level functions including file-system–like access to key-value storage for small chunks of data, a notification-based callback mechanism, and rudimentary access control. On the one hand, this approach has several benefits: Based on this low-level functionality, more complex services and data structures for the coordination of application processes (e. g., distributed queues) can be implemented. Furthermore, the fact that state-of-the-art coordination services are replicated frees application developers from the need to deal with fault-tolerance–related problems, as the coordination service does not represent a single point of failure. On the other hand, this flexibility comes at a price: With more complex services being implemented at the coordination-service client (i. e., as part of the distributed application), reusability is limited and maintenance becomes more difficult. In addition, there is a performance overhead for cases in which a complex operation requires multiple remote calls to the coordination service. We have seen such problem in the course of implementing FT-BPEL that externalizes the coordination of the active replication of businesses process to a coordination service (i.e., ZooKeeper). As we show in our evaluation, this problem gets worse the more application processes access a coordination service concurrently.

To address the disadvantages of current coordination services and improve the the performance of FT-BPEL, we propose an *extendable coordination service*. In contrast to existing solutions, in our approach, more complex services and data structures are not implemented at the client side but within modules ("*extensions*") that are executed at the servers running the coordination service. As a result, implementations of coordination service clients can be greatly simplified. In fact, in most usage scenarios, only a single remote call to the coordination service is required.

In our service, an extension is realized as a sequence of regular coordination service operations that are processed atomically. This way, an extension can benefit from the flexibility offered by the low-level API of a regular coordination service while achieving good performance under contention.

Besides enhancing the implementation of abstractions already used by current distributed applications, extensions also allow programmers to introduce new features that cannot be provided based on the functionality of traditional coordination services: By registering custom extensions, for example, it is possible to integrate assertions into our extendable coordination service that perform sanity checks on input data, improving protection against faulty clients. Furthermore, extensions may be used to execute automatic conversion routines for legacy clients, supporting scenarios in which the format of the coordination-related data managed on behalf of an application differs across program versions.

In particular, this chapter makes the following three contributions: First, it proposes a coordination service whose functionality can be enhanced dynamically by introducing customized extensions (see Section 7.3). Second, it provides details on our prototype of an extendable coordination service based on ZooKeeper [HKJR10], a coordination middleware widely used in industry (see Section 7.4). Third, it presents two case studies, a priority queue and a quota-enforcement service, illustrating both the flexibility and efficiency of our approach (see Section 7.5). In addition, Section 7.2 gives background on state-of-the-art coordination services, Section 7.6 discusses related work, and Section 7.7 concludes.

## 7.2 Background

This section provides background information on the basic functionality of a coordination service and presents an example of a higher-level abstraction built on top of it.

### 7.2.1 Coordination Services

Despite their differences in detail, coordination services like Chubby [Bur06] and ZooKeeper [HKJR10] expose a similar API to the client (i. e., a process of a distributed application, see Figure 7.1). Information is stored in *nodes* which can be created (`create`[1]) and deleted (`delete`) by a client. Furthermore, there are operations to store (`setData`) and retrieve (`getData`) the data assigned to a node. In general, there are two different types of nodes: *ephemeral nodes* are automatically deleted when the session of the client who created the node ends (e. g., due to a fault); in contrast, *regular nodes* persist after the end of a client session.

Besides managing data, current coordination services provide a callback mechanism to inform clients about certain events including, for example, the creation or deletion of a node, or the modification of the data assigned to a node (see Figure 7.1). On the occurrence of an event a client has registered a *watch* for, the coordination service performs a callback notifying the client about the event. Using this functionality, a client is, for example, able to implement failure detection of another client by setting a deletion watch on an ephemeral node created by the client to monitor.

[1] Note that we use the ZooKeeper terms here as our prototype is based on this particular coordination service.

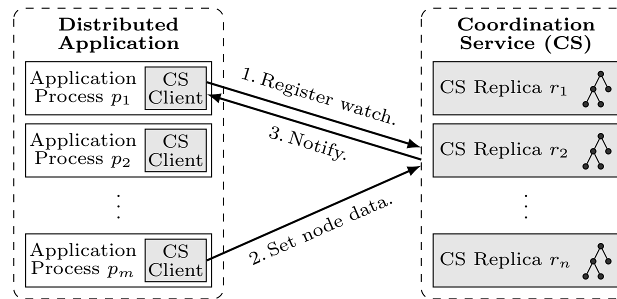



Figure 7.1: Callback mechanism usage example: An application process $p_1$ registers a data watch on a node; when the node's data is updated, the coordination service notifies $p_1$ about the modification.

### 7.2.2 Usage Example: Priority Queue

Based on the low-level API provided by the coordination service, application programmers can implement more complex data structures to be used for the coordination of processes. Figure 7.2 shows an example implementation of a distributed priority queue (derived from the queue implementation in [Zoo]) that can be applied to exchange data between two processes running on different machines: a *producer* and a *consumer*. New elements are added to the queue by the producer calling `insert`; the element with the highest priority is dequeued by the consumer calling `remove`.

To insert an element `b` into the queue, the producer creates a new node and sets its data[2] to `b` (L. 8). The priority `p` of the element is thereby encoded in the node name by appending `p` to a default name prefix (L. 5). To remove the head element of the queue, the consumer queries the coordination service to get the names of all nodes matching the default name prefix (L. 13). From the result set of node names, the consumer then locally determines the head of the queue by selecting the node name indicating the highest priority (L. 14). Knowing the head node, the consumer is able to retrieve its data from the coordination service (L. 17) before removing the node from the queue (L. 18).

Note that the priority-queue implementation in Figure 7.2 has two major drawbacks: First, while the `insert` operation involves only a single remote call to the coordination service (L. 8), the `remove` operation requires three remote calls (L. 13, 17, and 18), resulting in additional latency. Second, the implementation does not scale for multiple consumer processes: In order to prevent different consumers from returning the same element, entire `remove` operations would either have to be executed sequentially (which is difficult to achieve when consumer processes run on different machines) or they would have to be implemented optimistically; that is, if the `delete` call (L. 18) aborts due to a concurrent `remove` operation already having deleted the designated head node, a consumer must retry its `remove` (omitted in Figure 7.2). In Section 7.5.1, we show that the performance of the optimistic variant suffers from contention when multiple consumer processes access the queue concurrently.

## 7.3 Enhancing Coordination

The priority-queue example discussed in Section 7.2.2 illustrates the main disadvantage of state-of-the art coordination services: With implementations of higher-level data structures and ser-

[2]The ZooKeeper API allows a client to assign data to a node at creation time. Otherwise an additional `setData` call would be necessary for setting the node data.

vices being a composition of multiple low-level remote calls to the coordination service, performance and scalability become a major concern. We address this issue with an *extendable coordination service* that provides means to implement additional functionality directly at the server.

### 7.3.1 Basic Approach

To add functionality to our coordination service, programmers write *extensions* that are integrated via software modules. Depending on the mechanism an extension operates on, we distinguish between the following three types:

- During integration, a **node extension** registers a *virtual node* through which the extension will be accessible to the client. In contrast to a regular node, client operations invoked on a virtual node (or one of its sub nodes) are not directly executed by the coordination-service logic; instead, such requests are intercepted and redirected to the corresponding node extension.
- A **watch extension** may be used to customize/overwrite the behavior of the coordination service for a certain watch. Such an extension is executed each time a watch event of the corresponding type occurs.
- A **session extension** is triggered at creation and termination of a client session and is therefore suitable to perform initialization and cleanup tasks.

Note that an extension module providing additional functionality may be a composition of multiple extensions of possibly different types.

In general, an extension is free to use the entire API provided by the coordination service. As a consequence, a stateful extension, for example, is allowed to create own regular nodes to manage its internal state. Furthermore, a complex node extension, for example, may translate

```
1  CoordinationService cs = establish connection;

3  void insert(byte[] b, Priority p) {
4    /* Encode priority in node name. */
5    String nodeName = "/node-" + p;

7    /* Create node and set its data to b. */
8    cs.create(nodeName, b);
9  }

11 byte[] remove() {
12   /* Find the node with the highest priority. */
13   String[] nodes = get node names from cs;
14   String head = node from nodes
            with highest priority according to its name;

16   /* Get node data and remove node. */
17   byte[] b = cs.getData(head);
18   cs.delete(head);
19   return b;
20 }
```

Figure 7.2: Pseudo-code implementation of a priority-queue client (ZooKeeper): an element is represented by a node, the priority is encoded in the node name.

an incoming client request into a composite request comprising a sequence of low-level operations. Note that, in such a case, our coordination service guarantees that low-level operations belonging to the same composite request will be executed atomically (see Section 7.4.3).

### 7.3.2 Usage Example: Enhanced Priority Queue

Figure 7.3 shows how the implementation of the priority-queue client from Figure 7.2 can be greatly simplified by realizing the queue as a node extension that is accessed via a virtual node `/queue`. In contrast to the traditional implementation presented in Section 7.2.2, our extension variant only requires a single remote call for the removal of the head element from the queue.

When a client inserts an element into the queue by creating a sub node of `/queue` (L. C5), the request is forwarded to the queue extension, which in turn processes it without any modifications (L. E5); that is, the extension creates the sub node as a regular node. To dequeue the head element, a client issues a `getData` call to a (non-existent) sub node `/queue/next` (L. C10). On the reception of a `getData` call to this particular sub-node name, the extension removes the head element and returns its data to the client (L. E10-E17).

Although the steps executed during the dequeuing of the head element are identical to the corresponding procedure in the traditional priority-queue implementation (L. 12-19 in Figure 7.2), there is an important difference: the calls for learning the node names of queue elements (L. E11), for retrieving the data of the head element (L. E15), and for deleting the head-element node (L. E16) are all local calls with low performance overhead. Furthermore, with these three calls being processed atomically, the implementation does not suffer from contention, as shown in Section 7.5.1.

## 7.4 Extendable ZooKeeper

In this section, we present details on the implementation of *Extendable ZooKeeper (EZK)*, our prototype of an extendable coordination service, which is based on ZooKeeper [HKJR10].

### 7.4.1 Overview

EZK relies on actively-replicated ZooKeeper for fault tolerance. At the server side, EZK (like ZooKeeper) distinguishes between client requests that modify the state of the coordination service (e. g., by creating a node) and read-only client requests that do not (e. g., as they only read the data of a node). A read-only request is only executed on the server replica that has received the request from the client. In contrast, to ensure strong consistency, a state-modifying request is distributed using an atomic broadcast protocol [JRS11] and then processed by all server replicas.

For EZK, we introduce an *extension manager* component into each server replica which is mainly responsible for redirecting the control and data flow to the extensions registered. The extension manager performs different tasks for different types of extensions (see Section 7.3.1): On the reception of a client request, the extension manager checks whether the request accesses the virtual node of a node extension and, if this is the case, forwards the request to the corresponding extension. This way, a node extension is able to control the behavior of an incoming request before the request had any impact on the system. In addition, the extension manager intercepts watch events and, if available, redirects them to the watch extensions handling the

**Client Implementation**

```
C1  CoordinationService cs = establish connection;

C3  void insert(byte[] b, Priority p) {
C4    /* Create node and set its data to b. */
C5    cs.create("/queue/node-" + p, b);
C6  }

C8  byte[] remove() {
C9    /* Remove head node and return its data. */
C10   return cs.getData("/queue/next");
C11 }
```

**Coordination Service Extension Implementation**

```
E1  CoordinationServiceState local = local state;

E3  void create(String name, byte[] b) {
E4    /* Process request without modifications. */
E5    local.create(name, b);
E6  }

E8  byte[] getData(String name) {
E9    if("/queue/next".equals(name)) {
E10     /* Find node with the highest priority. */
E11     String[] nodes = get node names from local;
E12     String head = node from nodes
            with highest priority according to its name;

E14     /* Get node data and remove node. */
E15     byte[] b = local.getData(head);
E16     local.delete(head);
E17     return b;
E18   } else {
E19     /* Return data of regular node. */
E20     return local.getData(name);
E21   }
E22 }
```

Figure 7.3: Pseudo-code implementation of a priority queue in our extendable coordination service: the extension is represented by a virtual node `/queue`.

specific events, allowing the extension to customize the callback to the client. Finally, the extension manager also monitors ZooKeeper's session tracker and notifies the session extensions registered about the start and end of client sessions.

### 7.4.2 Managing an Extension

For extension management in EZK we provide a *built-in management extension* that is accessible through a virtual node `/extensions`. To register a custom extension, a client creates a sub node of `/extensions` and assigns all necessary configuration information as data to this management node. For a node extension, for example, the configuration information includes the name of the virtual node through which the extension can be used by a client, and a Java archive containing the extension code to execute when a request accesses this virtual node. Furthermore, a client is able to provide an *ephemeral* flag indicating whether the extension should be automatically removed by EZK when the session of the client who registered the extension ends; apart from that, an extension can always be removed by explicitly deleting its

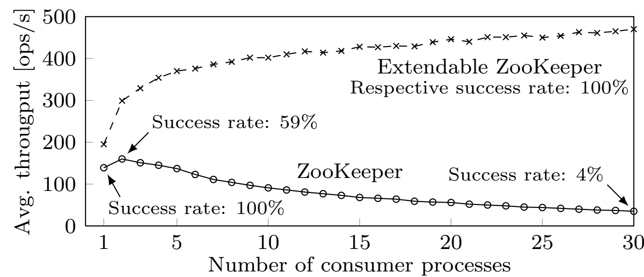



Figure 7.4: Throughput (i. e., successful dequeue operations) for different priority-queue implementations for different numbers of consumer processes.

corresponding management node.

When EZK's management extension receives a request from a client to register an extension, it verifies that the extension code submitted is a valid Java archive, and then distributes the request to all server replicas. By treating the request like any other state-modifying request, EZK ensures that all server replicas register the extension in a consistent manner. After registration is complete the extension manager starts to make use of the extension.

### 7.4.3 Atomic Execution of an Extension

Traditional implementations of complex operations comprising multiple remote calls to the coordination service (as, for example, removing the head element of a priority queue, see Section 7.2.2) require the state they operate on not to change between individual calls. As a consequence, such an operation may be aborted when two clients modify the same node concurrently, resulting in a significant performance penalty (see Section 7.5.1). We address this problem in EZK by executing complex operations atomically.

In ZooKeeper, each client request modifying the state of the coordination service is translated into a corresponding *transaction* which is then processed by all server replicas. In the default implementation a single state-modifying request leads to a single transaction. To support more complex operations, we introduce a new type of transaction in EZK, the *container transaction*, which may comprise a batch of multiple regular transactions. EZK guarantees that all transactions belonging to the same container transaction will be executed atomically on all server replicas, without interfering with other transactions. By including all transactions of the same extension-based operation in the same container transaction, EZK prevents concurrent state changes during the execution of an extension.

## 7.5 Case Studies

In this section, we evaluate the priority-queue extension introduced in Section 7.3.2. Furthermore, we present an additional example of how extensions can be used in our coordination service to efficiently provide more complex functionality. All experiments are conducted using a coordination-service cell comprising five server replicas (i. e., a typical configuration for ZooKeeper), each running in a virtual machine in Amazon EC2 [ec2]; coordination-service clients are executed in an additional virtual machine. As in practice distributed applications usually run in the same data center as the coordination service they rely on [Bur06], we allocate all virtual machines in the same EC2 region (i. e., Europe).

```
1 CoordinationService cs = establish connection;

3 void allocate(int amount) {
4   do {
5     /* Determine free quota and node version. */
6     (int free, int version) = cs.getData("/memory");

8     /* Retry if there is not enough quota. */
9     if(free < amount) sleep and continue;

11    /* Calculate and try to set new free quota. */
12    cs.setData("/memory", free - amount, version);
13  } while(setData call aborted unsuccessfully);
14 }
```

Figure 7.5: Pseudo-code implementation of a quota-server client in ZooKeeper: the current amount of free quota is stored in the data of `/memory`; to release quota, `allocate` is called with a negative `amount`.

### 7.5.1 Priority Queue

Our first case study compares a traditional priority-queue implementation (see Section 7.2.2) against our extension-based EZK variant (see Section 7.3.2). For both implementations, we measure the number of successful dequeue operations per second for a varying number of consumer processes accessing the queue concurrently. At all times during the experiments, we ensure that there are enough producer processes to prevent the queue from running empty. As a result, no dequeue operation will fail due to lack of items to remove.

Figure 7.4 presents the results of the experiments: For a single consumer process, the priority queues achieve an average throughput of 139 (ZooKeeper variant) and 195 (EZK) dequeue operations per second, respectively. The difference in performance is due to the fact that in the ZooKeeper implementation the `remove` operation comprises three (i. e., two read-only and one state-modifying) remote calls to the coordination service, whereas the extension-based EZK variant requires only a single (state-modifying) remote call.

Our results also show that for multiple consumer processes the ZooKeeper priority queue suffers from contention: Due to its optimistic approach a dequeue operation may be aborted when issued concurrently with another dequeue operation (see Section 7.2.2), causing the success rate to decrease for an increasing number of consumers. In contrast, dequeue operations in our EZK implementation are executed atomically and therefore always succeed on a non-empty queue. As a result, the extension-based EZK variant achieves better scalability than the traditional priority queue.

### 7.5.2 Quota Enforcement Service

Our second case study is a fault-tolerant quota enforcement service guaranteeing upper bounds for the overall resource usage (e. g., number of CPUs, memory usage, network bandwidth) of a distributed application [BDK12]. In order to enforce a global quota, each time an application process wants to dynamically allocate additional resources, it is required to ask the quota service for permission. The quota service only grants this permission in case the combined resource usage of all processes of the application does not exceed a certain threshold; otherwise the allocation request is declined and the application process is required to wait until additional free quota becomes available, for example, due to another process terminating and therefore releasing its resources.

### Traditional Implementation

Figure 7.5 illustrates how to implement a quota service based on a state-of-the-art coordination service. In this approach, information about free resource quotas (in the example: the amount of free memory available) is stored in the data assigned to a resource-specific node (i. e., `/memory`). To request permission for using additional quota, an application process invokes the quota client's `allocate` function indicating the amount of quota to be allocated (L. 3). Due to the traditional coordination service only providing functionality to get and set the data assigned to a node, but lacking means to modify node data based on its current value, the quota client needs to split up the operation into three steps: First, the client retrieves the data assigned to `/memory` (L. 6), thereby learning the application's current amount of free quota. Next, the quota client checks whether the application has enough free quota available to grant the permission (L. 9). If this is the case, the client locally computes the new amount of free quota and updates the corresponding node data at the coordination service (L. 12).

Note that the optimistic procedure described above is only correct as long as the data assigned to `/memory` does not change between the `getData` (L. 6) and `setData` (L. 12) remote calls. However, as different quota clients could invoke `allocate` for the same resource type concurrently, this condition may not always be justified. To address this problem, state-of-the-art coordination services like Chubby [Bur06] and ZooKeeper [HKJR10] use node-specific version counters (which are incremented each time the data of a node is reassigned) to provide a `setData` operation with compare-and-swap semantics. Such an operation only succeeds if the current version matches an expected value (L. 12), in this case, the version number that corresponds to the contents the quota client has retrieved (L. 6). If the two version numbers differ, the `setData` operation fails and the quota client retries the entire allocation procedure (L. 13).

### Extension-based Implementation

In contrast to the traditional implementation presented in Section 7.5.2 where remote calls issued by a quota client may be aborted due to contention, allocation requests in our extension-based EZK variant of the quota enforcement service (see Figure 7.6) are always granted when enough free quota is available. Here, to issue an allocation request, a client invokes a `setData` call to the virtual `/memory-quota` node passing the amount of quota to allocate as data (L. C6). In the absence of network and server faults, this call only aborts if the amount requested exceeds the free quota currently available (L. E8), in which case the quota client retries the procedure (L. C7) after a certain period of time (omitted in Figure 7.6). At the EZK server, the quota enforcement extension functions as a proxy for a regular node `/memory`: For each incoming `setData` call to this particular node (L. E4), the extension translates the request into a sequence of operations (i. e., a read (L. E5), a check (L. E8), and an update (L. E11)) that are processed atomically.

### Evaluation

We evaluate both implementations of the quota enforcement service varying the number of quota clients accessing the service concurrently from 1 to 40. During a test run, each client repeatedly requests 100 quota units, and when the quota is granted (possibly after multiple retries), immediately releases it again. In all cases, the total amount of quota available is limited to 1500 units. As a consequence, in scenarios with more than 15 concurrent quota clients, allocation requests may be aborted due to lack of free quota.

**Client Implementation**

```
C1  CoordinationService cs = establish connection;

C3  void allocate(int amount) {
C4    do {
C5      /* Issue quota demand. */
C6      cs.setData("/memory-quota", amount);
C7    } while(setData call aborted unsuccessfully);
C8  }
```

**Coordination Service Extension Implementation**

```
E1  CoordinationServiceState local = local state;

E3  void setData(String name, int amount) {
E4    if("/memory-quota".equals(name)) {
E5      int free = local.getData("/memory");

E7      /* Abort if there is not enough quota. */
E8      if(free < amount) abort;

E10     /* Calculate and set new free quota. */
E11     local.setData("/memory", free - amount);
E12   } else {
E13     /* Set data of regular node. */
E14     local.setData(name, amount);
E15   }
E16 }
```

Figure 7.6: Pseudo-code implementation of a quota server in our extendable coordination service: a call to `setData` only aborts if there is not enough quota.

The throughput results for this experiment presented in Figure 7.7 show that our EZK quota server provides better scalability than the state-of-the-art ZooKeeper variant. For a small number of concurrent clients, the fact that the total amount of quota is limited has no effect: As in the priority-queue experiment (see Section 7.5.1), the ZooKeeper implementation suffers from contention, whereas the throughput of the EZK quota server improves for multiple quota clients. For more than 15 quota clients, the fraction of aborted allocation requests increases in both implementations with every additional client, leading to an observable throughput decrease for the EZK quota server for more than 20 clients.

Figure 7.5.2 shows that the costs for a single quota allocation greatly differ between both quota-service implementations: For 40 clients, due to contention and the limited amount of total quota, it takes a ZooKeeper client more than 57 remote calls to the coordination service to be granted the quota requested; an EZK quota client on average has to issue less than 2 remote calls for the same scenario. Note that in the ZooKeeper variant, release operations are also subject to contention, requiring up to 28 remote calls per successful operation. In contrast, the release operation in our EZK implementation always succeeds using a single remote call.

## 7.6 Related Work

With the advent of large distributed file systems emerged the need to coordinate read and write accesses on different nodes. This problem was solved by distributed lock managers [ST87], the predecessors of current coordination services.

In contrast to the file-system–oriented coordination middleware systems Chubby [Bur06]

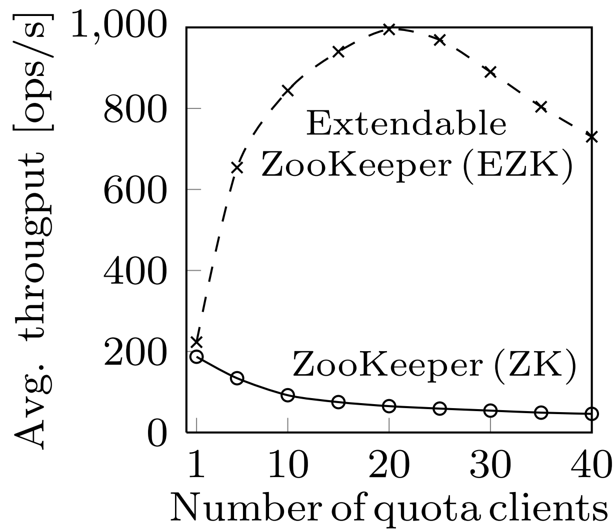



Figure 7.7: Throughput (i. e., successful allocation and release operations) for different quota-server variants; the total quota is limited to the demand of 15 clients.

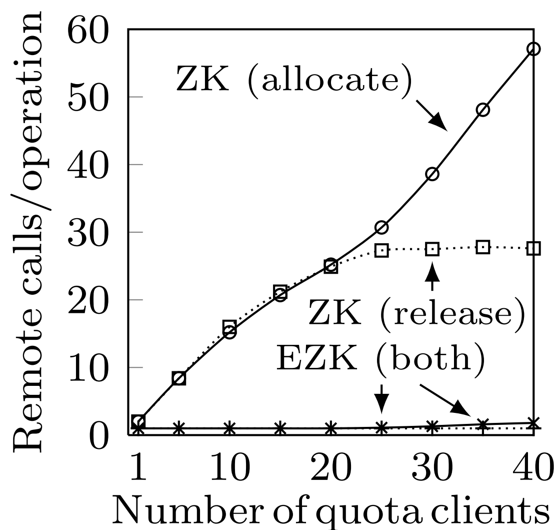



Figure 7.8: Costs (i. e., remote calls per operation) for different quota-server variants; the total quota is limited to the demand of 15 clients.

and ZooKeeper [HKJR10], DepSpace [BACF08] is a Byzantine fault-tolerant coordination service which implements the tuple space model. As the tuple space abstraction does not provide an operation to alter stored tuples, in order to update the data associated with a tuple, the tuple has to be removed from the tuple space, modified, and reinserted afterwards. In consequence, implementations of high-level data structures and services built over DepSpace are expected to also suffer from contention for multiple concurrent clients. Note that, with our approach not being limited to a specific interface, this problem could be addressed by an extension-based variant of DepSpace.

Boxwood [MMN+04] shares our goal of freeing application developers from the need to deal with issues like consistency, dependability, or efficiency of complex high-level abstractions. However, unlike our work, Boxwood focuses on storage infrastructure, not coordination middleware systems. In addition, the set of abstractions and services exposed by Boxwood is static, whereas our extendable coordination service allows clients to dynamically customize the behavior of existing operations and/or introduce entirely new functionality.

Relational database management systems rely on stored procedures [SAH87] (i. e., compositions of multiple SQL statements) to reduce network traffic between applications and the database, similar to our use of extensions to minimize the number of remote calls a client has to issue to the coordination service. In active database systems [PD99], triggers (i. e., a special form of stored procedures) can be registered to handle certain events, for example, the insertion, modification, or deletion of a record. As such, triggers are related to watches in coordination services. The main difference is that in general a trigger is a database-specific mechanism which is transparent to applications. As a result, applications are not able to change the behavior of a trigger. In contrast, our extendable coordination service offers applications the flexibility to customize the service using a composition of extensions operating on nodes, watches, and sessions.

## 7.7 Conclusion

This chapter proposed to enhance the coordination of distributed applications by relying on an extendable coordination service. It alleviates shortcomings of the current FT-BPEL implementation and allows programmers to dynamically introduce custom high-level abstractions which are then executed on the server side. Our evaluation shows that by processing complex operations atomically, an extendable coordination service offers significantly better performance and scalability than state-of-the-art implementations.

# Bibliography

[AAD+93] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.

[ABF08] Eduardo Adilio Pelinson Alchieri, Alysson Neves Bessani, and Joni da Silva Fraga. A dependable infrastructure for cooperative web services coordination. In *Proceedings of the 2008 IEEE International Conference on Web Services*, pages 21–28, 2008.

[ABFG08] Eduardo Adilio Pelinson Alchieri, Alysson Neves Bessani, Joni Silva Fraga, and Fabiola Greve. Byzantine consensus with unknown participants. In *Proceedings of the 12th International Conference on Principles of Distributed Systems – OPODIS'08*, pages 22–40, 2008.

[ABND95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[ACKL08] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Byzantine replication under attack. In *Proceedings of the IEEE/IFIP 38th International Conference on Dependable Systems and Networks*, pages 197–206, June 2008.

[ACKM06] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk Paxos: Optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

[ADD+10] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Steward: Scaling Byzantine fault-tolerant replication to wide area networks. *IEEE Transactions on Dependable and Secure Computing*, 7:80–93, 2010.

[AEMGG+05] Michael Abd-El-Malek, Gregory Ganger, Garth Goodson, Michael Reiter, and Jay Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 59–74, October 2005.

[AKMS11] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *Journal of the ACM*, 58:7:1–7:32, April 2011.

[AL03] Paul C. Attie and Nancy A. Lynch. Dynamic input/output automata: a formal model for dynamic systems. Technical Report MIT-LCS-TR-902, MIT Laboratory for Computer Science, Cambridge, MA, USA, July 2003.

[ALM+10] Eric Anderson, Xiaozhou Li, Arif Merchant, Mehul A. Shah, Kevin Smathers, Joseph Tucek, Mustafa Uysal, and Jay J. Wylie. Efficient eventual consistency

in Pahoehoe, an erasure-coded key-blob archive. In *Proceedings of the 40th International Conference on Dependable Systems and Networks (DSN-DCCS)*, 2010.

[ALPW10] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. RACS: a case for cloud storage diversity. In *Symposium on Cloud Computing (SoCC)*, pages 229–240, 2010.

[amaa] Amazon Auto Scalling. http://aws.amazon.com/autoscaling/, retrieved Sep. 11, 2012.

[Amab] Amazon S3 availability event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html, retrieved Dec. 6, 2011.

[Amac] Amazon gets 'black eye' from cloud outage. http://www.computerworld.com/s/article/9216064/Amazon_gets_black_eye_from_cloud_outage, retrieved Dec. 6, 2011.

[Amad] Amazon Simple Storage Service. http://aws.amazon.com/s3/, retrieved Dec. 6, 2011.

[Ama10] Amazon S3 FAQ: What data consistency model does amazon S3 employ? http://aws.amazon.com/s3/faqs/, 2010.

[AMM05] Sergio Andreozzi, Danilo Montesi, and Rocco Moretti. Xmatch: A language for satisfaction-based selection of grid services. *Sci. Program.*, 13(4):299–316, October 2005.

[Apa] Apache HBase. http://hbase.apache.org/.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004.

[BACF08] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Fraga. DepSpace: A Byzantine fault-tolerant coordination service. In *Proceedings of the EuroSys 2008 Conference (EuroSys '08)*, pages 163–176, 2008.

[BCE+12] Cristina Băsescu, Christian Cachin, Ittay Eyal, Robert Haas, Alessandro Sorniotti, Marko Vukolić, and Ido Zachevsky. Robust data sharing with key-value stores. In *Proc. Intl. Conference on Dependable Systems and Networks (DSN'12)*, June 2012.

[BCQ+11] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. In *6th ACM SIGOPS/EuroSys European Systems Conference (EuroSys'11)*. ACM, Apr 2011.

[BDK12] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. DQMP: A decentralized protocol to enforce global quotas in cloud environments. In *Proceedings of the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS '12)*, 2012.

[BFG+08] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In *Proc. of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 251–264, 2008.

[BJO09] Kevin D. Bowers, Ari Juels, and Alina Oprea. HAIL: a high-availability and integrity layer for cloud storage. In *Proc. of the 16th ACM Conference on Computer and Communications Security - CCS'09*, pages 187–198, 2009.

[BPN+11] Sven Bugiel, Thomas Pöppelmann, Stefan Nürnberger, Ahmad-Reza Sadeghi, and Thomas Schneider. Amazonia: When elasticity snaps back. In *18th ACM Conference on Computer and Communications Security (CCS'11)*. ACM, Oct 2011.

[Bra84] Gabriel Bracha. An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing - PODC'84*, pages 154–162, August 1984.

[BSC+08] A. N. Bessani, P. Sousa, M. Correia, N. F. Neves, and P. Veríssimo. The CRUTIAL way of critical infrastructure protection. *IEEE Security & Privacy*, pages 44–51, Nov/Dec 2008.

[Bur06] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 335–350, 2006.

[Cac09] Christian Cachin. Yet another visit to Paxos. Technical report, IBM Research Zurich, 2009.

[CBPS10] Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer, 2010.

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 205–218, 2006.

[CGKV09] Gregory Chockler, Rachid Guerraoui, Idit Keidar, and Marko Vukolić. Reliable distributed storage. *IEEE Computer*, 42(4):60–67, 2009.

[CGR11] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming (Second Edition)*. Springer, 2011.

[CHV10] Christian Cachin, Robert Haas, and Marko Vukolić. Dependable services in the intercloud: Storage primer. Research Report RZ 3783, IBM Research, October 2010.

[CJS12] Christian Cachin, Birgit Junker, and Alessandro Sorniotti. On limitations of using cloud storage for data replication. In *Proc. 6th Workshop on Recent Advances in Intrusion Tolerance and reSilience (WRAITS 2012), DSN 2012 Workshops*, June 2012.

[CKPS01] Cristian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology: CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[CL02] Miguel Castro and Barbara Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4):398–461, 2002.

[CM02] Gregory Chockler and Dahlia Malkhi. Active disk Paxos with infinitely many processes. In *Proc. of the 21st Symposium on Principles of Distributed Computing – PODC'02*, pages 78–87, 2002.

[CM05] Gregory Chockler and Dahlia Malkhi. Active disk Paxos with infinitely many processes. *Distributed Computing*, 18:73–84, 2005.

[CML+06] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 177–190, November 2006.

[CNV06] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96, 2006.

[CPBC11] Pedro Costa, Marcelo Pasin, Alysson N. Bessani, and Miguel Correia. Byzantine fault-tolerant MapReduce: Faults are not just crashes. In *Proc. of the 3rd Int. Conference on Cloud Computing Technology and Science (CloudCom'11)*, 2011.

[CRS+11] Emanuele Cesena, Gianluca Ramunno, Roberto Sassu, Davide Vernizzi, and Antonio Lioy. On scalability of remote attestation. In *ACM STC'11: Proceedings of the 6th ACM Workshop on Scalable Trusted Computing*. ACM, October 2011. To appear.

[CT06] Christian Cachin and Stefano Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. In *Proceedings of the International Conference on Dependable Systems and Networks - DSN 2006*, pages 115–124, June 2006.

[CWA+09] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design & Implementation*, April 2009.

[CWO+11] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 143–157, New York, NY, USA, 2011. ACM.

[DCGN03] M. Dahlin, B. Chandra, L. Gao, and A. Nayate. End-to-end WAN service availability. *ACM/IEEE Transactions on Networking*, 11(2), April 2003.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 137–150, December 2004.

[DGG05] Assia Doudou, Benoit Garbinato, and Rachid Guerraoui. *Dependable Computing Systems Paradigms, Performance Issues, and Applications*, chapter Tolerating Arbitrary Failures with State Machine Replication, pages 27–56. Wiley, 2005.

[DGLV10] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Marko Vukolic. Fast access to distributed atomic memory. *SIAM Journal on Computing*, 39(8):3752–3783, 2010.

[DGV05] P. Dutta, R. Guerraoui, and M. Vukolic. Best-case complexity of asynchronous Byzantine consensus. Technical Report 200499, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2005.

[DH08] Danny Dolev and Ezra N. Hoch. Constant-space localized Byzantine consensus. In *Proc. of the 22nd Int. Symp. on Distributed Computing – DISC '08*, 2008.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Symposium on Operating System Principles (SOSP)*, pages 205–220, 2007.

[DK11] Tobias Distler and Rüdiger Kapitza. Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency. In *Proceedings of the EuroSys 2011 Conference (EuroSys'11)*, pages 91–105, 2011.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[ec2] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2/, retrieved Sep. 11, 2012.

[Ehs10] UK NHS Systems and Services. http://www.connectingforhealth.nhs.uk/, 2010.

[ES00] Burkhard Englert and Alexander A. Shvartsman. Graceful quorum reconfiguration in a robust emulation of shared memory. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 454–463, 2000.

[f W11] f Web Services. Amazon simple storage service faqs. http://aws.amazon.com/s3/faqs/, 2011.

[FC09] Jorge Fox and Siobhán Clarke. Exploring approaches to dynamic adaptation. In *Proceedings of the 3rd International DiscCoTec Workshop on Middleware-Application Interaction*, MAI '09, pages 19–24, New York, NY, USA, 2009. ACM.

[FZFF10] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, pages 337–350, October 2010.

[GBG+11] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro. OS diversity for intrusion tolerance: Myth or reality? In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems and Networks*, pages 383 – 394, Hong Kong, China, June 2011.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proc. of the 19th ACM Symposium on Operating Systems Principles – SOSP'03*, pages 29–43, 2003.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *Symposium on Operating System Principles (SOSP)*, pages 150–162, 1979.

[GKQV10] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *Proc. of the 5th European Conf. on Computer systems – EuroSys'10*, 2010.

[GL03] Eli Gafni and Leslie Lamport. Disk Paxos. *Distributed Computing*, 16(1):1–20, January 2003.

[GLK+10] Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. Comet: an active distributed key-value store. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[GLS10] Seth Gilbert, Nancy Lynch, and Alexander Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23:225–272, 2010.

[Gma] Gmail back soon for everyone. http://gmailblog.blogspot.com/2011/02/gmail-back-soon-for-everyone.html, retrieved Dec. 6, 2011.

[GNA+98] Garth Gibson, David Nagle, Khalil Amiri, Jeff Butler, Fay Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proc. of the 8th Int. Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS'98*, pages 92–103, 1998.

[GNS09] Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Fault-tolerant semifast implementations of atomic read/write registers. *Journal of Parallel Distributed Computing*, 69(1):62–79, 2009.

[goG] GOGRID. http://www.gogrid.com/, retrieved Sep. 11, 2012.

[Gre10] Melvin Greer. Survivability and information assurance in the cloud. In *Proc. of the 4th Workshop on Recent Advances in Intrusion-Tolerant Systems – WRAITS'10*, June 2010.

[GWGR04] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Micheal K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of Dependable Systems and Networks - DSN 2004*, pages 135–144, June 2004.

[Ham07] James Hamilton. On designing and deploying Internet-scale services. In *Proc. of the 21st Large Installation System Administration Conference – LISA'07*, pages 231–242, 2007.

[HDS+11] Michael Hanley, Tyler Dean, Will Schroeder, Matt Houy, Randall F. Trzeciak, and Joji Montelibano. An analysis of technical observations in insider theft of intellectual property cases. Technical Note CMU/SEI-2011-TN-006, Carnegie Mellon Software Engineering Institute, February 2011.

[Hen09] Alyssa Henry. Cloud storage FUD (failure, uncertainty, and durability). Keynote Address at the 7th USENIX Conference on File and Storage Technologies, February 2009.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[HGR07] James Hendricks, Gregory Ganger, and Michael Reiter. Low-overhead byzantine fault-tolerant storage. In *Proc. of the 21st ACM Symposium on Operating Systems Principles – SOSP'07*, pages 73–86, 2007.

[HKJR10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC '10)*, pages 145–158, 2010.

[HLM03] Maurice Herlihy, Victor Lucangco, and Mark Moir. Obstruction-free syncronization: double-ended queues as an example. In *Proc. of 23th IEEE International Conference on Distributed Computing Systems - ICDCS 2003*, July 2003.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programmming Languages and Systems*, 12(3):463–492, July 1990.

[ibm] IBM SmartCloud. http://www.ibm.com/cloud-computing/us/en/, retrieved Sep. 11, 2012.

[Inv07] Paola Inverardi. Software of the future is the future of software? In *Proceedings of the 2nd international conference on Trustworthy global computing*, TGC'06, pages 69–85, Berlin, Heidelberg, 2007. Springer-Verlag.

[jcl] jclouds — multi-cloud library. http://www.jclouds.org/, retrieved December 6, 2011.

[JCT98] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, May 1998.

[JHJ+10] Gueyoung Jung, M.A. Hiltunen, K.R. Joshi, R.D. Schlichting, and C. Pu. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 62 –73, june 2010.

[JRS11] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 41st International Conference on Dependable Systems and Networks (DSN '11)*, pages 245–256, 2011.

[KAD07a] Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin. SafeStore: A durable and practical storage system. In *Proceedings of the USENIX Annual Technical Conference - USENIX'07*, 2007.

[KAD+07b] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proc. of the 21st ACM Symposium on Operating Systems Principles - SOSP'07*, October 2007.

[Kra93] Hugo Krawczyk. Secret sharing made short. In *Proc. of the 13th Int. Cryptology Conference – CRYPTO'93*, pages 136–146, August 1993.

[KVB11] Bernhard Kauer, Paulo Verissimo, and Alysson Bessani. Recursive virtual machines for advanced security mechanisms. In *1st Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments (DCDV'11)*, 2011.

[LAB+06] Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The smart way to migrate replicated stateful services. *SIGOPS Oper. Syst. Rev.*, 40(4):103–115, April 2006.

[Lam74] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Lam86] Leslie Lamport. On interprocess communication (part II). *Distributed Computing*, 1(1):203–213, January 1986.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions Computer Systems*, 16(2):133–169, May 1998.

[Lam01] Butler Lampson. The ABCD's of Paxos. In *Proceedings of the 20th annual ACM Symposium on Principles of Distributed Computing*, page 13, 2001.

[LaS10] LaSIGE/Navigators. BFT-SMaRt: High-performance Byzantine-Fault-Tolerant State Machine Replication. *http://code.google.com/p/bft-smart*, 2010.

[LM07] Jinyuan Li and David Mazieres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *Proceedings of 4th Symposium on Networked Systems Design and Implementation - NSDI 2007*, April 2007.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Operating Systems Review*, 44:35–40, 2010.

[LR06] Barbara Liskov and Rodrigo Rodrigues. Tolerating Byzantine faulty clients in a quorum system. In *Proc. of 26th IEEE Int. Conf. on Distributed Computing Systems - ICDCS 2006*, 2006.

[LS97] Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing (FTCS)*, pages 272–281, 1997.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, July 1982.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[MA05] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine consensus. In *Proceedings of the Dependable Systems and Networks - DSN 2005*, pages 402–411, June 2005.

[MAD02] J. P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proceedings of the 16th International Conference on Distributed Computing*, volume 2508 of *LNCS*, pages 311–325. Springer-Verlag, October 2002.

[May10] Mike May. Forecast calls for clouds over biological computing. *Nature Medicine*, 16(6), 2010.

[Met09] Cade Metz. DDoS attack rains down on Amazon cloud. *The Register*, October 2009. http://www.theregister.co.uk/2009/10/05/amazon_bitbucket_outage/.

[Mez] Mezeo: Cloud storage platform. http://www.mezeo.com/, retrieved Dec. 6, 2011.

[MG11] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. In *Reports on Computer Systems Technology*. NIST, Sep. 2011. Special Publication 800-145.

[MHS11] Zarko Milosevic, Martin Hutle, and André Schiper. On the reduction of atomic broadcast to consensus with Byzantine faults. In *Proc. of the 30th IEEE Int. Symp. on Reliable Distributed Systems – SRDS'11*, 2011.

[Mil92] David L. Mills. Network time protocol (version 3): Specification, implementation and analysis. IETF RFC 1305, March 1992.

[MJWS10] John C. McCullough, JohnDunagan, Alec Wolman, and Alex C. Snoeren. Stout: An adaptive interface to scalable cloud storage. In *Proc. of the USENIX Annual Technical Conference – ATC 2010*, pages 47–60, June 2010.

[MMN+04] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 105–120, 2004.

[mos] mOSAIC Cloud: Open source api and platform for multiple clouds. http://www.mosaic-cloud.eu/, retrieved Sep. 11, 2012.

[MR97] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *Proceedings of the 29th annual ACM Symposium on Theory of Computing*, pages 569–578, 1997.

[MR98] Dahlia Malkhi and Michael Reiter. Secure and scalable replication in Phalanx. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems - SRDS'98*, pages 51–60, October 1998.

[MRMS10] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo Seltzer. Provenance for the cloud. In *Proc. of the 8th USENIX Conference on File and Storage Technologies – FAST'10*, pages 197–210, 2010.

[MSKC04] Philip. K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. A taxonomy of compositional adaptation. Technical report, Michigan State University, East Lansing, Michigan, 2004.

[MSL+10] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. In *Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation – OSDI 2010*, pages 307–322, October 2010.

[MTJ+08] John Maccormick, Chandramohan A. Thekkath, Marcus Jager, Kristof Roomp, Lidong Zhou, and Ryan Peterson. Niobe: A practical replication protocol. *ACM Transactions on Storage*, 3:1:1–1:43, February 2008.

[Nao09] Erica Naone. Are we safeguarding social data? Technology Review published by MIT Review, http://www.technologyreview.com/blog/editors/22924/, February 2009.

[nic] NICTA. http://www.nicta.com.au/, retrieved Sep. 11, 2012.

[occ] Open Cloud Computing Interface. http://occi-wg.org/, retrieved Sep. 11, 2012.

[OCJ08] Jon Oberheide, Evan Cooke, and Farnam Jahanian. Exploiting Live Virtual Machine Migration. In *BlackHat DC Briefings*, Washington DC, February 2008.

[OL88] Brian M. Oki and Barbara Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 8–17, 1988.

[PD99] Norman W. Paton and Oscar Díaz. Active database systems. *ACM Computing Surveys*, 31(1):63–103, 1999.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proc. of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, 1988.

[Pla07] James S. Plank. Jerasure: A library in C/C++ facilitating erasure coding for storage applications. Technical Report CS-07-603, University of Tennessee, September 2007.

[Rab89] Michael Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, February 1989.

[raca] Rackspace. http://www.rackspace.com/, retrieved Sep. 11, 2012.

[racb] Rackspace hosting. http://www.rackspacecloud.com/cloud_hosting_products/files/, retrieved Dec. 6, 2011.

[Rap11] J.R. Raphael. The 10 worst cloud outages (and what we can learn from them). Infoworld. Available at http://www.infoworld.com/d/cloud-computing/the-10-worst-cloud-outages-and-what-we-can-learn-them-902, 2011.

[RBL+09] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Cáceres, M. Ben-Yehuda, W. Emmerich, and F. Galán. The reservoir model and architecture for open federated cloud computing. *IBM J. Res. Dev.*, 53(4):535–545, July 2009.

[Rei11] Hans P. Reiser. Byzantine Fault Tolerance for the Cloud. In *Workshop on Cryptography and Security in Clouds*. IBM, Zurich, Mar 2011. http://www.zurich.ibm.com/ cca/csc2011/program.html.

[RK07] Hans Reiser and Rudiger Kapitza. VM-FIT: Supporting intrusion tolerance with virtualisation technology. In *Proceedings of the 1st Workshop on Recent Advances in Intrusion-Tolerant Systems*, pages 18–22, June 2007.

[RLS98] R. Raman, M. Livny, and M. Solomon. Matchmaking: distributed resource management for high throughput computing. In *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, pages 140 –146, jul 1998.

[RMS10] Olivier Rütti, Zarko Milosevic, and André Schiper. Generic construction of consensus algorithms for benign and Byzantine faults. In *Proc. of the Int. Conf. on Dependable Systems and Networks - DSN'10*, 2010.

[RP11] Jason K. Resch and James S. Plank. AONT-RS: Blending security and performance in dispersed storage systems. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[SAH87] Michael Stonebraker, Jeff Anton, and Eric Hanson. Extending a database system with procedures. *Transactions on Database Systems*, 12(3):350–376, 1987.

[Sar09] David Sarno. Microsoft says lost sidekick data will be restored to users. *Los Angeles Times*, Oct. 15th 2009.

[SB12] João Sousa and Alysson Bessani. From byzantine consensus to bft state machine replication: A latency-optimal transformation. In *Proc. of the 9th European Conference on Dependable Computing (EDCC'12)*, May 2012.

[SBC+10] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, 2010.

[SCC+10] Alexander Shraer, Christian Cachin, Asaf Cidon, Idit Keidar, Yan Michalevsky, and Dani Shaket. Venus: Verification for untrusted cloud storage. In *Proc. of the ACM Cloud Computing Security Workshop – CCSW'10*, 2010.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22, December 1990.

[Sch99] Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology - CRYPTO'99*, pages 148–164, August 1999.

[SG07] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1), 2007.

[SGMV07] Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti. Potshards: Secure long-term storage without encryption. In *Proc. of the USENIX Annual Technical Conference – ATC 2007*, pages 143–156, June 2007.

[Sha79] Adi Shamir. How to share a secret. *Communications of ACM*, 22(11):612–613, November 1979.

[SPW03] Cheng Shao, Evelyn Pierce, and Jennifer L. Welch. Multi-writer consistency conditions for shared memory objects. In *Distributed Computing (DISC)*, pages 106–120, 2003.

[SRG12] Jyoti Shetty, Anala M R, and Shobha G. Article: A survey on techniques of secure live migration of virtual machine. *International Journal of Computer Applications*, 39(12):34–39, February 2012. Published by Foundation of Computer Science, New York, USA.

[SRMJ12] Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Junqueira. Dynamic reconfiguration of primary/backup clusters. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 39–39, Berkeley, CA, USA, 2012. USENIX Association.

[SSSS11] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. A cost-aware elasticity provisioning system for the cloud. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 559 –570, june 2011.

[ST87] W. Snaman and D. Thiel. The VAX/VMS distributed lock manager. *Digital Technical Journal*, 5:29–44, 1987.

[TDP+94] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proc. of the 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149, September 1994.

[VA86] Paul M. B. Vitányi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware (detailed abstract). In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 233–243, 1986.

[VBP12] Paulo Verissimo, Alysson Bessani, and Marcelo Pasin. The tclouds architecture: Open and resilient cloud-of-clouds computing. In *2nd International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments (DCDV'12)*, 2012.

[VCB+11] Giuliana Veronese, Miguel Correia, Alysson Bessani, Lau Lung, and Paulo Verissimo. Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, 2011.

[VCBL09] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 135–144, 2009.

[VCBL10] Giuliana Veronese, Miguel Correia, Alysson Bessani, and Lau Lung. Ebawa: Efficient byzantine agreement for wide-area networks. In *12th IEEE International High Assurance Systems Engineering Symposium (HASE'10)*, 2010.

[VNC03] P. Veríssimo, N. F. Neves, and M. Correia. Intrusion-tolerant architectures: Concepts and design. In R. Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*, pages 3–36. Springer-Verlag, 2003.

[VNC+06] P. Verissimo, N. F. Neves, C. Cachin, J. Poritz, D. Powell, Y. Deswarte, R. Stroud, and I. Welch. Intrusion-tolerant middleware: The road to automatic security. *IEEE Security and Privacy*, 4(4):54–62, Jul./Aug. 2006.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[Vol] Voldemort: A distributed database. `http://project-voldemort.com/`, retrieved Dec. 6, 2011.

[VSV09] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. Cumulus: Filesystem backup to the cloud. *ACM Transactions on Storage*, 5(4):1–28, 2009.

[VSV12] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. BlueSky: A cloud-backed file system for the enterprise. In *Proc. of the 10th USENIX Conference on File and Storage Technologies – FAST'12*, 2012.

[Vuk10] Marko Vukolić. The Byzantine empire in the intercloud. *ACM SIGACT News*, 41:105–111, September 2010.

[WBM+06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 307–320, 2006.

[win] Windows Azure. http://www.windowsazure.com, retrieved Sep. 11, 2012.

[yur] Yuruware. http://www.yuruware.com/, retrieved Sep. 11, 2012.

[YXYB10] Yunqi Ye, Liangliang Xiao, I-Ling Yen, and Farokh Bastani. Secure, dependable, and high performance cloud storage. In *Proceedings of the 29th Symposium on Reliable Distributed Systems (SRDS)*, pages 194–203, 2010.

[Zie04] Piotr Zielinski. Paxos at war. Technical Report UCAM-CL-TR-593, University of Cambridge, Computer Laboratory, 2004.

[Zoo] ZooKeeper Tutorial: Queues. http://wiki.apache.org/hadoop/ZooKeeper/Tutorial.

# Appendix A

# Additional Material for Chapter 4

## A.1 Auxiliary functions

This section presents the two auxiliary functions used in the protocols of Algorithms 1 and 2. The description of these functions is available in Table 4.1.

**Algorithm 13:** DepSky-A and DepSky-CA auxiliary functions.

```
1  function queryMetadata(du)
2  begin
3      m[0 .. n − 1] ⟵ ⊥
4      parallel for 0 ≤ i ≤ n − 1 do
5          tmp_i ⟵ cloud_i.get(du, "metadata")
6          if verify(tmp_i, K^{du}_{uw}) then m[i] ⟵ tmp_i
7      wait until |{i : m[i] ≠ ⊥}| ≥ n − f
8      for 0 ≤ i ≤ n − 1 do cloud_i.cancel_pending()
9      return m
10 procedure writeQuorum(du, name, value)
11 begin
12     ok[0 .. n − 1] ⟵ false
13     parallel for 0 ≤ i ≤ n − 1 do
14         ok[i] ⟵ cloud_i.put(du, name[i], value[i])
15     wait until |{i : ok[i] = true}| ≥ n − f
16     for 0 ≤ i ≤ n − 1 do cloud_i.cancel_pending()
```

The two functions presented in Algorithm 13 are similar and equally simple: the process just accesses all the $n$ clouds in parallel to get or put data and waits for replies from a quorum of clouds, canceling non-answered RPCs.

## A.2 Storage Protocols Correctness

This section presents correctness proofs of the DepSky-A and DepSky-CA protocols. The first lemma states that the auxiliary functions presented in the previous section are wait-free.

**Lemma 1.** *A correct process will not block executing writeQuorum or queryMetadata.*

*Proof.* Both operations require $n - f$ clouds to answer the put and get requests. For *writeQuorum*, these replies are just *acks* and they will always be received since at most $f$ clouds are

faulty. For the *queryMetadata*, the operation is finished only if one metadata file is available. Since we are considering only non-malicious writers, a metadata written in a cloud is always valid and thus correctly signed using $K^{du}_{rw}$. It means that a valid metadata file will be read from at least $n - f$ clouds and the process will choose one of these files and finish the algorithm. □

The next two lemmas state that if a correctly signed metadata is obtained from the cloud providers (using *queryMetadata*) the corresponding data can also be retrieved and that the metadata stored on DEPSKY-A and DEPSKY-CA satisfy the properties of a regular register [Lam86] (if the clouds provide this consistency semantics).

**Lemma 2.** *The value associated with the metadata with greatest version number returned by* $queryMetadata$*, from now on called* outstanding metadata*, is available on at least* $f + 1$ *non-faulty clouds.*

*Proof.* Recall that only valid metadata files will be returned by $queryMetadata$. These metadata will be written only by a non-malicious writer that follows the DepSkyAWrite (resp. DepSkyCAWrite) protocol. In this protocol, the data value is written on a quorum of clouds on line 8 (resp. line 14) of Algorithm 1 (resp. Algorithm 2), and then the metadata is generated and written on a quorum of clouds on lines 9-12 (resp. lines 15-18). Consequently, a metadata is only put on a cloud if its associated value was already put on a quorum of clouds. It implies that if a metadata is read, its associated value was already written on $n - f$ clouds, from which at least $n - f - f \geq f + 1$ are correct. □

**Lemma 3.** *The outstanding metadata obtained on an DepSkyARead (resp. DepSkyCARead) concurrent with zero or more DepSkyAWrites (resp. DepSkyCAWrites) is the metadata written on the last complete write or being written by one of the concurrent writes.*

*Proof.* Assuming that a client reads an outstanding metadata $m$, we have to show that $m$ was written on the last complete write or is being written concurrently with the read. This proof can easily be obtained by contradiction. Suppose $m$ was written before the start of the last complete write before the read and that it was the metadata with greatest version number returned from $queryMetadata$. This is clearly impossible because $m$ was overwritten by the last complete write (which has a greater version number) and thus will never be selected as the outstanding metadata. It means that $m$ can only correspond to the last complete write or to a write being executed concurrently with the read. □

With the previous lemmas we can prove the wait-freedom of the DEPSKY-A and DEPSKY-CA registers, showing that their operations will never block.

**Theorem 7.** *All* DEPSKY *read and write operations are wait-free operations.*

*Proof.* Both Algorithms 1 and 2 use functions $queryMetadata$ and $writeQuorum$. As shown in Lemma 1, these operations can not block. Besides that, read operations make processes wait for the value associated with the outstanding metadata. Lemma 2 states that there are at least $f + 1$ correct clouds with this data, and thus at least one of them will answer the RPC of lines 21 and 27 of Algorithms 1 and 2, respectively, with values that match the digest contained on the metadata (or the different block digests in the case of DEPSKY-CA) and make $d[i] \neq \perp$, releasing itself from the barrier and completing the algorithm. □

The next two theorems show that DEPSKY-A and DEPSKY-CA implement single-writer multi-reader regular registers.

**Theorem 8.** *A client reading a* DEPSKY-A *register in parallel with zero or more writes (by the same writer) will read the last complete write or one of the values being written.*

*Proof.* Lemma 3 states that the outstanding metadata obtained on lines 16-17 of Algorithm 1 corresponds to the last write executed or one of the writes being executed. Lemma 2 states that the value associated with this metadata is available from at least $f + 1$ correct clouds, thus it can be obtained by the client on lines 20-24: just a single valid reply will suffice for releasing the barrier of line 24 and return the value. □

**Theorem 9.** *A client reading a* DEPSKY-CA *register in parallel with zero or more writes (by the same writer) will read the last complete write or one of the values being written.*

*Proof.* This proof is similar to the one for DEPSKY-A. Lemma 3 states that the outstanding metadata obtained on lines 22-23 of Algorithm 2 corresponds to the last write executed or one of the writes being executed concurrently. Lemma 2 states that the values associated with this metadata are stored on at least $f + 1$ non-faulty clouds, thus a reader can obtain them through the execution of lines 26-30: all non-faulty clouds will return their values corresponding to the outstanding metadata allowing the reader to decode the encrypted value, combine the key shares and decrypt the read data (lines 33-35), inverting the processing done by the writer on DepSkyCAWrite (lines 7-10). □

## A.3 Lock Protocol Correctness

This section presents correctness proofs for the data unit locking protocol. Recall from Section 4.4.1 that this protocol requires two extra assumptions: (1) contending writers have their clocks synchronized with precision $\Delta/2$ and (2) the storage clouds provides at least read-after-write consistency.

Before the main proofs, we need to present a basic lemma that shows that a lock file created in a quorum of clouds is read in a later listing of files from a quorum of clouds.

**Lemma 4.** *An object $o$ created with the operation $writeQuorum(du, o, v)$ and not removed will appear in at least one result of later $list(du)$ operations executed on a quorum of clouds.*

*Proof.* The $writeQuorum(du, o, v)$ operation is only completed when the object is created/written in a quorum of at least $n - f$ clouds (line 15 of Algorithm 13). If a client tries to list the objects of $du$ on a quorum of clouds, at least one of the $n - f$ clouds will provide it since there is at least one correct cloud between any two quoruns ($(n - f) + (n - f) - n > f$). □

In order to prove the mutual exclusion on lock possession we need to precisely define what it means for a process to hold the lock for a given data unit.

**Definition 7.** *A correct client $c$ is said to* hold the write lock for a $du$ at a given time $t$ *if an object du-lock-c-T containing $sign(\text{du-lock-c-}T, K_c)$ with $T + \Delta < t$ appears in at least one $list(du)$ result when this operation is executed in a quorum of clouds.*

With this definition, we can prove the safety and liveness properties of Algorithm 3.

**Theorem 10** (Mutual exclusion)**.** *At any given time $t$, there is at most one correct client that holds the lock for a data unit $du$.*

*Proof.* Assume this is false: there is a time $t$ in which two correct clients $c_1$ and $c_2$ hold the lock for $du$. We will prove that this assumption leads to a contradiction.

If both $c_1$ and $c_2$ hold the write lock for $du$ we have that both du-lock-$c_1$-$T_1$ and du-lock-$c_2$-$T_2$ with $T_1 + \Delta < t$ and $T_2 + \Delta < t$ are returned in $list(du)$ operations from a quorum of clouds. Algorithm 3 and Lemma 4 states that it can only happens if both $c_1$ and $c_2$ wrote valid lock files (line 12) and did not remove them (line 19). In order for this to happen, both $c_1$ and $c_2$ must see only their lock files in their second $list(du)$ on the clouds (lines 14-16).

Two situations may arise when $c_1$ and $c_2$ acquire write locks for $du$: either $c_1$ (resp. $c_2$) writes its lock file before $c_2$ (resp. $c_1$) lists the lock files the second time (i.e., either $w_1$ precedes $r_2$ or $w_2$ precedes $r_1$) or one's lock file is being written while the other is listing lock files for the second time (i.e., either $w_1$ is executed concurrently with $r_2$ or $w_2$ is executed concurrently with $r_1$).

In the first situation, when $c_2$ (resp. $c_1$) lists available locks, it will see both lock files and thus remove du-lock-$c_2$-$T_2$ (resp. du-lock-$c_1$-$T_1$), releasing the lock (lines 14-20).

The second situation is more complicated because now we have to analyze the start and finish of each phase of the algorithm. Consider the case in which $c_1$ finishes writing its lock file (line 12) after $c_2$ executes the second list (lines 14-15). Clearly, in this case $c_2$ may or may not see du-lock-$c_1$-$T_1$ in line 18. However, we can say that the second list of $c_1$ will see du-lock-$c_2$-$T_2$ since it is executed after $c_1$ lock file is written, which happens, only after $c_2$ start its second list, and consequently after its lock file write. It means that the condition of line 18 will be true for $c_1$, and it will remove du-lock-$c_1$-$T_1$, releasing its lock. The symmetric case ($c_2$ finishes writing its lock after $c_1$ executes the second list) also holds.

In both situations we have a contradiction, i.e., it is impossible to have an execution and time in which two correct clients hold the lock for $du$. □

**Theorem 11** (Obstruction-freedom). *If a correct client tries to obtain the lock for a data unit $du$ without contention it will succeed.*

*Proof.* When there is no other valid lock in the cloud (i.e., the condition of line 10 holds), $c$ will write du-lock-$c$-$T$ on a quorum of clouds. This lock file will be the only valid lock file read on the second list (the condition of line 18 will not hold) since (1.) no other valid lock file is available on the clouds, (2.) no other client is trying to acquire the lock, and (3.) Lemma 4 states that if the lock file was written it will be read. After this, $c$ acquire the lock and return it. □

## A.4 Consistency Proportionality

In this section we prove the consistency proportionality of the DEPSKY-A and DEPSKY-CA protocols considering some popular consistency models [Lam86, TDP+94, Vog09].

In the following theorem we designate by the *weakest cloud* the correct cloud that provides less guarantees in terms of consistency. In homogeneous environments, all clouds will provide the same consistency, but in heterogeneous environments other clouds will provide at least the guarantees of the weakest cloud.

**Theorem 12.** *If the weakest cloud used in a* DEPSKY-CA *setup satisfies a consistency model $\mathcal{C}$, the data unit provided by* DEPSKY-CA *also satisfies $\mathcal{C}$ for any $\mathcal{C} \in$* {eventual, read-your-writes, monotonic reads, writes-follow-reads, monotonic writes, read-after-write}.

*(Sketch).* Notice that cloud consistency issues only affect metadata readings since in the DEPSKY-CA (Algorithms 2), after the $max_id$ variable is defined (line 23), the clients keep reading the clouds until the data value is read (lines 24-32). So, even with eventual consistency (the weakest guarantee we consider), if the metadata file pointing to the last version is read, the data will eventually be read.

Let $Q_w$ be the quorum of clouds in which the metadata of the last executed write $w$ was written and let $Q_r$ be the quorum of clouds where $queryMetadata$ obtained an array of $n - f$ metadata files on a posterior read $r$. Let $cloud \in Q_w \cap Q_r$ be the *weakest cloud* among the available clouds. For each of the considered consistency models, we will prove that if $cloud$ provides this consistency, the register implemented by DEPSKY-CA provides the same consistency.

For *eventual consistency* [Vog09], if the outstanding metadata file was written on $cloud$ and no other write operation is executed, it will be eventually available for reading in this cloud, and then its associated data will be fetched from the clouds. As a consequence, the data described in the metadata file will be read eventually, satisfying this model.

For *read-your-writes* consistency, if both $w$ and $r$ are executed by $c$, the fact that $cloud$ provides this consistency means that at least this cloud will return the metadata written in $w$ during $r$ execution. Consequently, the result of the read will be the value written in $w$, satisfying this model.

For *monotonic reads* consistency, assume $c$ executed $r$ and also another posterior read $r'$. Let $Q_{r'}$ be the quorum accessed when reading metadata file on $r'$. Let $cloud' \in Q_r \cap Q_{r'}$ be a *correct* cloud providing *monotonic reads* consistency. We have to prove that $r'$ will return the same data of $r$ or a value written in a posterior write. Since $cloud'$ satisfy monotonic reads, the metadata file read in $r'$ will be either the one read in $r$ or another one written by a posterior write. In any of the two cases, the corresponding value returned will satisfy the monotonic reads consistency.

For *writes-follow-reads* consistency, the result is trivial since, as long as we have no contending writers, the metadata files are written with increasing version numbers. Since the clouds provide at least this consistency, it is impossible to observe (and to propagate) writes in a different order they were executed, i.e., to observe them in an order different from the version numbers of their metadata.

The same arguments holds for *monotonic writes*: since the clouds provide at least this consistency, it is impossible to observe the writes in a different order they were executed.

Finally, for *read-after-write* consistency[1], the safety properties proved in previous section (see Theorem 9) can be easily generalized for any read-after-write model. If the outstanding metadata file was written on a $cloud$ satisfying this consistency model during write $w$ and no other write operation is executed, any read succeeding $w$ will see this file, and its associated data will be fetched from the cloud. □

Since the critical step of Theorem 12' proof uses the intersection between metadata' reads and writes, the following corollary states that the result just proved for DEPSKY-CA is also valid for DEPSKY-A. The key reason is that both protocols read and validate metadata files in the same way, as can be seen in lines 16-17 of Algorithm 1 and 22-23 of Algorithm 2).

**Corollary 13.** *If the weakest cloud used in a* DEPSKY-A *setup satisfies a consistency model* $\mathcal{C}$*, the data unit provided by* DEPSKY-A *also satisfies* $\mathcal{C}$ *for any* $\mathcal{C} \in$ {eventual, read-your-writes, monotonic reads, writes-follow-reads, monotonic writes, read-after-write}.

[1] Any of the consistency models introduced in [Lam86] satisfies *read-after-write* since the reads return the value written in the last complete operation in absence of contention.

# Appendix B

# Proofs for Chapter 5

## B.1 Correctness of the Register Constructions

This section proves the correctness of the register constructions presented in Section 5.3.1.

For establishing the correctness of the algorithms, note first that every client accesses the KVS objects in a well-formed manner, as ensured by the corresponding checks in Algorithm 6 (line 4), Algorithm 8 (lines 4 and 10), and Algorithm 9 (lines 4 and 8).

A global execution of the system consists of invocations and responses of two kinds: those of the emulated register and those of the KVS base objects. In order to distinguish between them, we let $\bar{\sigma}$ denote an execution of the register (with **read** and **write** operations) and let $\sigma$ denote an execution of the KVS base objects (with **put**, **get**, **list**, and **remove** operations).

We say that a KVS-operation $o$ is *induced* by a register operation $\bar{o}$ when the client executing $\bar{o}$ invoked $o$ according to its algorithm for executing $\bar{o}$. Furthermore, a **read** operation *reads a version ver* when the returned value has been associated with *ver* (Algorithm 6 line 7), and a **write** operation *writes a version ver* when an induced **put** operation stores a value under a temporary key corresponding to *ver* (Algorithm 8 line 11).

At a high level, the register emulations are correct because the **read** and **write** operations always access a majority of the KVSs, and hence every two operations access at least one common KVS. Furthermore, each KVS stores two copies of a value under the eternal and under temporary keys. Because the algorithm for reading is carefully adjusted to the garbage-collection routine, every **read** operation returns a legitimate value in finite time. Section B.1.1 below makes this argument precise for the regular register, and Section B.1.2 addresses the atomic register.

### B.1.1 MRMW-Regular Register

We prove safety (Theorem 14) and liveness (Theorem 15) for the emulation of the MWMR-regular register. Consider any execution $\bar{\sigma}$ of the algorithm, the induced execution $\sigma$ of the KVSs, and a real-time sequential permutation $\pi$ of $\sigma$ (note that $\sigma$ is determined by the operations on the atomic KVSs). Let $\pi_i$ denote the sequence of actions from $\pi$ that occur at some KVS replica $i$.

According to Algorithm 8, every **write** operation to the register induces exactly two **put** operations, one with a temporary key and one with the eternal key; the **write** may also remove some temporary keys. We first establish that for every KVS, the maximum of all versions that correspond to an associated temporary key always increases.

**Lemma 5** (KVS version monotonicity)**.** *Consider a KVS $i$, a write operation $w$ that writes version ver, and some operation* **put**$_i$ *in $\pi_i$ induced by $w$ with a temporary key. Then the response*

*of any operation* $\mathbf{list}_i$ *in* $\pi_i$ *that follows* $\mathbf{put}_i$ *contains at least one temporary key that corresponds to a version equal to or larger than ver.*

*Proof.* We show this by induction on the length of some prefix of $\pi_i$ that is followed by an imaginary **list**′ operation. (Note that **list** does not modify the state of KVS $i$.)

Initially, no versions have been written, and the claim is vacuously true for the empty prefix. According to the induction assumption, the claim holds for some prefix $\rho_i$. We argue that it also holds for every extension of $\rho_i$. When $\rho_i$ is extended by a $\mathbf{put}_i$ operation, the claim still holds. Indeed, the claim can only be affected when $\rho_i$ is extended by an operation $\mathbf{remove}_i$ with a key that corresponds to version *ver and* when no $\mathbf{put}_i$ operation with a temporary key that corresponds to a larger version than *ver* exists in $\rho_i$.

A $\mathbf{remove}_i$ operation is executed by some client that executes a **write** operation and function **putInKVS** in two cases. In the first case, when Algorithm 7 invokes operation $\mathbf{remove}_i$ in line 5, it has previously executed $\mathbf{list}_i$ and excluded from *obsolete* the temporary key corresponding to the largest version *ver*′. The induction assumption implies that $ver' \geq ver$. Hence, there exists a temporary key corresponding to $ver' \geq ver$ also after $\mathbf{remove}_i$ completes.

In the second case, when Algorithm 7 invokes $\mathbf{remove}_i$ in line 9, then it has already stored a temporary key corresponding to a larger version than *ver* through operation $\mathbf{put}_i$ (line 8), according to the algorithm. The claim follows. □

**Lemma 6** (Partial order)**.** *In an execution* $\bar{\sigma}$ *of the algorithm, the versions of the read and write operations in* $\bar{\sigma}$ *respect the partial order of the operations in* $\bar{\sigma}$*:*

*a) When a* ***write*** *operation* $w$ *writes a version* $v_w$ *and a subsequent (in* $\bar{\sigma}$*)* ***read*** *operation* $r$ *reads a version* $v_r$*, then* $v_w \leq v_r$*.*

*b) When a* ***write*** *operation* $w_1$ *writes a version* $v_1$ *and a subsequent* ***write*** *operation* $w_2$ *writes a version* $v_2$*, then* $v_1 < v_2$*.*

*Proof.* For part a), note that both operations return only after receiving responses from a majority of KVSs. Suppose KVS $i$ belongs to the majority accessed by the **putInKVS** function during $w$ and to the majority accessed by $r$. Since $w \prec_{\bar{\sigma}} r$, the $\mathbf{put}_i$ operation induced by $w$ precedes the first $\mathbf{list}_i$ operation induced by $r$. Therefore, the latter returns at least one temporary key corresponding to a version that is $v_w$ or larger according to Lemma 5.

Consider now the execution of function **getFromKVS** (Algorithm 5) for KVS $i$. The previous statement shows that the client sets $v_0 \geq v_w$ in line 5. The function only returns a version that is at least $v_0$. As Algorithm 6 takes the maximal version returned from a KVS, the version $v_r$ of $r$ is not smaller than $v_w$.

The argument for the write operations in part b) is similar. Suppose that KVS $i$ belongs to the majority accessed by the **putInKVS** function during $w_1$ and to the majority accessed by the **list** operation during $w_2$. As $w_1 \prec_{\bar{\sigma}} w_2$, the $\mathbf{put}_i$ operation induced by $w_1$ precedes the $\mathbf{list}_i$ operation induced by $w_2$. Therefore, the latter returns at least one temporary key corresponding to a version that is $v_1$ or larger according to Lemma 5. Hence, the computed previous maximum version $\langle seq_{\max}, id_{\max} \rangle$ of Algorithm 8 in $w_2$ is at least $v_1$. Subsequently, operation $w_2$ at client $c$ determines its version $v_2 = \langle seq_{\max} + 1, c \rangle > \langle seq_{\max}, id_{\max} \rangle \geq v_1$. □

The two lemmas prepare the way for the following theorem. It shows that the emulation respects the specification of a multi-reader multi-writer regular register.

**Theorem 14** (MRMW-regular safety)**.** *Every well-formed execution* $\bar{\sigma}$ *of the MRMW-regular register emulation in Algorithms 6 and 8 is MRMW-regular.*

*Proof.* Note that a **read** only reads a version that was written by a **write** operation. We construct a sequential permutation $\bar{\pi}$ of $\bar{\sigma}$ by ordering all **write** operations of $\bar{\sigma}$ according to their versions and then adding all **read** operations after their matching **write** operation; concurrent **read** operations are added in arbitrary after, the others in the same order as in $\bar{\sigma}$.

Let $r$ be a **read** operation in $\bar{\sigma}$ and denote by $\bar{\sigma}_r$ and by $\bar{\pi}_r$ the subsequences of $\bar{\sigma}$ and $\bar{pi}$ according to Definition 3, respectively. They contain only $r$ and those **write** operations that do not follow $r$ in $\bar{\sigma}$. We show that $\bar{\pi}_r$ is a legal real-time sequential permutation of $\bar{\sigma}_r$.

Due to the construction of $\bar{\pi}$, operation $r$ returns the value written by the last preceding **write** operation or $\bot$ if there is no such **write**. The sequence $\bar{\pi}_r$ is therefore legal with respect to a register.

It remains to show that $\bar{\pi}_r$ respects the real-time order of $\bar{\sigma}_r$. Consider two operations $o_1$ and $o_2$ in $\bar{\sigma}_r$ such that $o_1 \prec_{\bar{\sigma}_r} o_2$. Hence, also $o_1 \prec_{\bar{\sigma}} o_2$. Note that $o_1$ and $o_2$ are either both **write** operations or $o_1$ is a **write** operation and $o_2$ is the **read** operation $r$. If $o_1$ is a **write** of a version $v_1$ and $o_2$ is a **write** of a version $v_2$, then Lemma 6a shows that $v_1 < v_2$. According to the construction of $\bar{\pi}$, we conclude that $o_1 \prec_{\bar{\pi}} o_2$. If $o_1$ is a **write** of a version $v_1$ and $o_2$ is a **read** of a version $v_2$, then Lemma 6b shows that $o_1 \prec_{\bar{\pi}} o_2$, again according to the construction of $\bar{\pi}$. By the construction of $\bar{\pi}_r$, this means that $o_1 \prec_{\bar{\pi}_r} o_2$. Hence, $\bar{\pi}_r$ is also a real-time sequential permutation of $\bar{\sigma}_r$. □

It remains to show that the register operations are also live. We first address the **read** operation, and subsequently the **write** operation.

**Lemma 7** (Wait-free read)**.** *Every **read** operation completes in finite time.*

*Proof.* The algorithm for reading (Algorithm 6) calls the function **getFromKVS** once for every KVS and completes after this call returns for a majority of the KVSs. As only a minority of KVSs may fail, it remains to show that when a client $c$ invokes **getFromKVS** for a correct KVS $i$, it returns in finite time.

Algorithm 5 implements **getFromKVS**. It first obtains a list *list* of all temporary keys from KVS $i$ and returns if no such key exists. If some temporary key is found, it determines the corresponding largest version $ver_0$ and enters a loop.

Towards a contradiction, assume that client $c$ never exits the loop in some execution $\bar{\sigma}$ and consider the induced execution $\sigma$ of the KVSs.

We examine one iteration of the loop. Note that since all operations of $c$ are wait-free, the iteration eventually terminates. Prior to starting the iteration, the client determines *list* from an operation $\textbf{list}_i$. In line 8 the algorithm attempts to retrieve the value associated with key $v_c = \max(list)$ through an operation $\textbf{get}_c(v_c)$. This returns FAIL and the client retrieves the eternal key with an operation $\textbf{get}_c(\text{ETERNAL})$. We observe that $\textbf{list}_c \prec_\sigma \textbf{get}_c(v_c) \prec_\sigma \textbf{get}_c(\text{ETERNAL})$.

Since $\textbf{get}_c(v_c)$ fails, some client must have removed it from the KVS with a $\textbf{remove}(v_c)$ operation. Applying Lemma 5 to version $v_c$ now implies that prior to the invocation of $\textbf{get}_c(v_c)$, there exists a temporary key in KVS $i$ corresponding to a version $v_d > v_c$ that was stored by a client $d$. Denote the operation that stored $v_d$ by $\textbf{put}_d(v_d)$. Combined with the previous observation, we conclude that

$$\textbf{list}_c \prec_\sigma \textbf{put}_d(v_d) \prec_\sigma \textbf{get}_c(v_c) \prec_\sigma \textbf{get}_c(\text{ETERNAL}). \tag{B.1}$$

Furthermore, according to Algorithm 7, client $d$ has stored a tuple containing $v_d > v_c$ under the eternal key prior to $\textbf{put}_d(v_d)$ with an operation $\textbf{put}_d(\text{ETERNAL})$. But the subsequent $\textbf{get}_c(\text{ETERNAL})$ by client $c$ returns a value containing a version *smaller* than $v_c$. Hence, there

must be an *extra* client $e$ writing concurrently, and its version-value pair has overwritten $v_d$ and the associated value under the eternal key. This means that operation $\mathbf{put}_e(\text{ETERNAL})$ precedes $\mathbf{get}_c(\text{ETERNAL})$ in $\sigma$ and stores a version $v_e < v_c$. Note that $\mathbf{put}_e(\text{ETERNAL})$ occurs exactly once for KVS $i$ during the write by $e$.

As client $e$ also uses Algorithm 8 for writing, its *results* variable must contain the responses of **list** operations from a majority of the KVSs. Denote by $\mathbf{list}_e$ its **list** operation whose response contains the largest version, as determined by $e$. Let $\mathbf{list}_c^0$ denote the initial list operation by $c$ that determined $ver_0$ in Algorithm 5 (line 5). We conclude that $\mathbf{list}_e$ precedes $\mathbf{list}_c^0$ in $\sigma$. Summarizing the partial-order constraints on $e$, we have

$$\mathbf{list}_e \prec_\sigma \mathbf{list}_c^0 \prec_\sigma \mathbf{put}_e(\text{ETERNAL}) \prec_\sigma \mathbf{get}_c(\text{ETERNAL}). \tag{B.2}$$

To conclude, in one iteration of the loop by reader $c$, some client $d$ concurrently writes to the register according to (B.1). An extra client $e$ concurrently writes as well and its **write** operation is invoked before $\mathbf{list}_c^0$ and irrevocably makes progress after $d$ invokes a **write** operation, according to (B.2). Therefore, client $e$ may cause *at most one* extra iteration of the loop by the reader. Since there are only a finite number of such clients, client $c$ eventually exits the loop. This contradicts the assumption that such an execution $\bar{\sigma}$ and the induced $\sigma$ exist, and the lemma follows. □

**Lemma 8** (Wait-free write)**.** *Every **write** operation completes in finite time.*

*Proof.* The algorithm for writing (Algorithm 8) calls the function **list** for every KVS, and continues after this call returns for a majority of the KVSs. Then, it calls the function **putInKVS** for every KVS and returns after this call returns for a majority of the KVSs. As only a minority of KVSs may fail, it remains to show that when a client $c$ invokes **putInKVS** for a correct KVS, it returns in finite time.

Algorithm 7 implements **putInKVS**. It calls **list**, possibly removes keys with **remove** and puts an eternal and possibly a temporary key in the KVS. Since all these operations are wait-free, the function returns in finite time. □

The next theorem summarizes these two lemmas and states that the emulation is wait-free.

**Theorem 15** (MRMW-regular liveness)**.** *Every **read** and **write** operation of the MRMW-regular register emulation in Algorithms 6 and 8 completes in finite time.*

### B.1.2 Atomic Register

We state the correctness theorems for the atomic register emulation and sketch their proofs. The complete proofs are similar to the ones for the MRMW-regular register emulation.

**Theorem 16** (Atomic safety)**.** *Every well-formed execution $\bar{\sigma}$ of the atomic register emulation in Algorithms 9 and 8 is atomic.*

*Proof sketch [ABND95].* Note that a **read** operation can only read a version that has been written by some **write** operation. We therefore construct a sequential permutation $\bar{\pi}$ by ordering the operations in $\bar{\sigma}$ according to their versions, placing all **read** operations immediately after the **write** operation with the same version. Two concurrent **read** operations in $\bar{\sigma}$ that read the same version may appear in arbitrary order; all other **read** operations appear ordered in the same way as in $\bar{\sigma}$.

We show that $\bar{\pi}$ is a legal real-time sequential permutation of $\bar{\sigma}$. From the construction of $\bar{\pi}$, it follows that every **read** operation returns the value written by the last preceding **write** operation, after which it was placed. Therefore, $\bar{\pi}$ is a legal sequence of operations with respect to a register.

It remains to show that $\bar{\pi}$ respects the real-time order of $\bar{\sigma}$. Consider two operations $o_1$ and $o_2$ in $\bar{\sigma}$ such that $o_1 \prec_{\bar{\sigma}} o_2$. Operation $o_1$ is either a **write** or a **read** operation. In both cases, it completes only after storing its (read or written) version $v_1$ together with its value at a majority of the KVSs under a temporary key that corresponds to $v_1$. Operation $o_2$ is either a **write** or a **read** operation. In both cases, it first lists the versions in a majority of the KVSs and determines the maximal version among the responses. Let this maximal version be $v_2$. Because at least one KVS lies in the intersection of the two sets accessed by $o_1$ and by $o_2$, we conclude that $v_2 \geq v_1$. If $o_2$ is a **read** operation, it reads version $v_2$, and if $o_2$ is a **write** operation, it writes a version strictly larger than $v_2$. Therefore, according to the construction of $\bar{\pi}$, we obtain $o_1 \prec_{\bar{\pi}} o_2$ as required. □

**Theorem 17** (Atomic liveness)**.** *Every **read** and **write** operation of the atomic register emulation in Algorithms 9 and 8 completes in finite time.*

*Proof sketch.* The only difference between the regular and the atomic register emulations lies in the write-back step at the end of the **atomicRead** function. It is easy to see that storing the temporary key corresponding to the same version again may only effect the algorithm and its analysis in a minor way. In particular, the argument for showing Lemma 7 must be extended to account for concurrent **read** operations, which may also store values to the KVSs now. Similar to a concurrent **write** operation, an atomic **read** operation may delay a reader by one iteration in its loop. But again, there are only a finite number of clients writing concurrently. A **read** operation therefore completes after a finite number of steps. □

## B.2 Analysis of the Efficiency of Register Constructions

**Theorem 18.** *The space complexity of the MRMW-regular register emulation at any KVS is at most two plus the point contention of concurrent write operations.*

*Proof.* Consider an execution $\bar{\sigma}$ of the MRMW-regular register emulation. We prove the theorem by considering the operations $o_1, o_2, \ldots$ of some legal real-time sequential permutation $\pi$ of $\sigma$, the KVS execution induced by $\bar{\sigma}$.

If at some operation $o_t$ the number of keys that is written to KVS $i$ but not removed is $x$, then at some operation prior to $o_t$, at least $x$ register operations were concurrently run. We prove by induction on $t$. Initially the claim holds since there are no keys put and no clients run. Assume it holds until $o_{t-1}$ and prove for $o_t$. If operation $o_t$ is not a **put**, then the number of put keys is the same as at $o_{t-1}$ and the claim holds by the induction assumption.

If operation $o_t$ is **put**$_i$, invoked by some client $c$, then it is performed by this client's **write**$_c$ that first removed all but one temporary keys in its GC routine (Algorithm 7 lines 4–9). These **remove** operations precede the **put** in $\bar{\sigma}$, and therefore also its real-time sequential permutation $\pi$. All (except maybe one) versions that were written by **write**s that completed before **write**$_c$ are therefore removed before operation $o_t$. The temporary keys in the system at $o_{t-1}$ are ones that were written by operations concurrent with **write**$_c$. The **put**$_c$ operation therefore increases their number by one, so the number of keys is at most the number of concurrent **write** operations, as required. □

**Theorem 19.** *In every emulation of a safe MRMW-register from KVS base objects, there exists some KVS with space complexity two.*

*Proof.* Toward a contradiction, suppose that every KVS stores only one key at any time.

Note that a client in an algorithm may access a KVS in an arbitrary way through the KVS interface. For modeling the limit on the number of stored values at a KVS, we assume that every **put** operation removes all previously stored keys and retains only the one stored by **put**. A client might still "compress" the content of a KVS by listing all keys, retrieving all stored values, and storing a representation of those values under one single key. In every emulation algorithm for the write operation, the client executes w.l.o.g. a "final" **put** operation on a KVS (if there is no such **put**, we add one at the end).

Note a client might also construct the key to be used in a **put** operation from values that it retrieved before. For instance, a client might store multiple values by simply using them as the key in put operations with empty values. This is allowed here and strengthens the lower bound. (Clearly, a practical KVS has a limit on the size of a key but the formal model does not.)

Since operations are executed asynchronously and can be delayed, a client may invoke an operation at some time, at some later time the object (KVS) executes the operation atomically, and again at some later time the client receives the response.

In every execution of an operation with more than $n/2$ correct KVSs it is possible that all operations of some client invoked on less than $n/2$ KVSs are delayed until after one or more client operations complete.

Consider now an execution with three KVSs, denoted $a$, $b$, and $c$. Consider three executions $\alpha$, $\beta$, and $\gamma$ that involve three clients $c_u$, $c_x$, and $c_r$.

**Execution $\alpha$.** Client $c_x$ invokes **write**$(x)$ and completes; let $T_\alpha^0$ be the point in time after that; suppose the final **put** operation from $c_x$ on KVS $b$ is delayed until after $T_\alpha^0$; then $b$ executes this **put**; let $T_\alpha^1$ be the time after that; suppose the corresponding response from $b$ to $c_x$ is delayed until the end of the execution.

Subsequently, after $T_\alpha^1$, client $c_r$ invokes **read** and completes with responses from $b$ and $c$; all operations from $c_r$ to $a$ are delayed until the end of the execution. Operation **read** returns $x$ according to the register specification.

**Execution $\beta$.** Client $c_x$ invokes **write**$(x)$ and completes, exactly as in $\alpha$; let $T_\beta^0$ $(= T_\alpha^0)$ be the time after that; suppose the final **put** operation from $c_x$ on KVS $b$ is delayed until the end of the execution.

Subsequently, after $T_\beta^0$, client $c_u$ invokes **write**$(u)$ and completes; let $T_\beta^1$ be the time after that; all operations from $c_u$ to KVS $c$ are delayed until the end of the execution.

Subsequently, after $T_\beta^1$, client $c_r$ invokes **read** and completes; all operations from $c_r$ to a are delayed until the end of the execution. Operation **read** by $c_r$ returns $u$ according to the register specification.

**Execution $\gamma$.** Client $c_x$ invokes **write**$(x)$ and completes, exactly as in $\beta$; let $T_\gamma^0$ $(= T_\beta^0)$ be the time after that; suppose the final **put** operation from $c_x$ to KVS $b$ is delayed until some later point in time.

Subsequently, after $T_\gamma^0$, client $c_u$ invokes **write**$(u)$ and completes, exactly as in $\beta$; let $T_\gamma^1$ $(= T_\beta^1)$ be the time after that; all operations from $c_u$ to KVS $c$ are delayed until the end of the execution.

Subsequently, after $T^1_\gamma$, the final **put** operation from $c_x$ to KVS $b$ induced by operation **write**$(x)$ is executed at KVS $b$; let $T^2_\gamma$ be the time after that; suppose the corresponding response from KVS $b$ to $c_x$ is delayed until the end of the execution.

Subsequently, after $T^2_\gamma$, client $c_r$ invokes **read** and completes; all operations from $c_r$ to KVS $a$ are delayed until the end of the execution. The **read** by $c_r$ returns $u$ by specification. But the states of KVSs $b$ and $c$ at $T^2_\gamma$ are the same as their states in $\alpha$ at $T^1_\alpha$, hence, $c_r$ returns $x$ as in $\alpha$, which contradicts the specification of the register. □

## B.3 Analysis of Cloud Storage Limitations

**Theorem 20.** *The consensus number of a replica object is infinite.*

*Proof.* We show how to implement a consensus object $C$ from a replica object $R$. The emulation is shown in Algorithm 2 and works as follows. At the start, the timestamp/value pair at $R$ is initialized to $(0, \perp)$. When a client invokes *decide*$(v)$ of $C$ with a proposal $v$, then the emulation tries to write the pair $(1, v)$ to $R$. Subsequently it reads the value stored by $R$ and returns it as the decision value.

**Algorithm 2** Implementation of a consensus object $C$ using a replica $R$

**operation** *decide*$(v)$
    $R$.*condwrite*$(1, v)$;
    $d \leftarrow R$.*read*$()$;
    **return** $d$;

The intuition behind this emulation is that only the first invocation of *condwrite* executed by $R$ will succeed in storing a value, say $v_1$, at $R$; it does not matter which client executes it. Every subsequent conditional write is simply ignored because $R$ already stores timestamp $1$. Once the first client has decided $v_1$, every other client that invokes *decide*$(v)$ also obtains $v_1$.

Object $C$ is wait-free because the implementation contains no loops, the underlying replica $R$ is wait-free, and every operation immediately returns. Furthermore, $C$ satisfies the *validity* property of a consensus object, because decided value is read from $R$ and because the proposal of at least one client is written to $R$ before it is read. Hence, the decided value must be an input from a client. Moreover, $C$ also implements *consistency* because only the first ever invocation of *condwrite* may change the value the is returned from $R$ by *read*. Hence, $C$ is a consensus object according to Definition 5.

Note that there is no upper bound on the number of clients that can write to or read from $R$; therefore, this implementation solves consensus for any number of clients. According to Definition 6, together with Theorem 1, this implies that the consensus number of a replica object is infinite. □

**Theorem 21.** *The consensus number of a key-value store object is one.*

*Proof.* We implement a KVS object $K$ with one atomic snapshot object $SO$. Recall that clients interact with every object in a well-formed manner.

The idea behind the emulation is to maintain in $SO[i]$ a list of all *put* and *remove* operations executed on the KVS by the client with index $i$. Every such operation is represented by a tuple $(i, ts, key, val) \in \{1, \dots, n\} \times \mathbb{N}_0 \times \mathcal{K} \times \mathcal{V}$, where $ts$ represents a logical timestamp that is incremented by every client when it executes an operation.

The history of operations of the client with index $i$, as stored in $D[i]$, consists of a concatenated list of such tuples:

$$D[i] \;=\; (i, ts, \mathit{key}, \mathit{val}) \parallel \cdots \parallel (i, ts', \mathit{key}', \mathit{val}').$$

For two tuples $\tau = (i, ts, \mathit{key}, \mathit{val})$ and $\tau' = (j, ts', \mathit{key}', \mathit{val}')$ we define a *tuple order* relation and say that $\tau$ is *bigger* than $\tau'$, denoted $\tau > \tau'$, whenever $ts > ts'$ or $ts = ts' \wedge i > j$.

We next describe the implementation; a formal statement appears in Algorithm 3.

The operation *put*$(\mathit{key}, \mathit{val})$ by a client with index $i$ first scans $SO$ and retrieves all histories. Then it determines the maximal tuple $(j, ts, k, v)$ from all histories according to tuple order. It increments the timestamp, sets $t \leftarrow ts + 1$, appends the tuple $(i, t, \mathit{key}, \mathit{val})$ to the list in $D[i]$, and uses the result to update its entry in $SO$. Recall that only the client with index $i$ may invoke *update*$(i, \cdot)$.

Operation *get*$(\mathit{key})$ just scans the snapshot object and searches in the returned histories for the largest tuple according to tuple order whose key equals *key*; denote this by $(i, ts, \mathit{key}, \mathit{val})$. If such a tuple is found, the operation returns *val*; otherwise, it returns $\perp$.

To execute *remove*$(\mathit{key})$, the implementation stores the special character $\perp$ under *key* using the *put* operation already implemented.

Finally, the *list*$()$ operation scans the snapshot object, examines every tuple from every history, and retains, for every key, the maximal tuple according to tuple order. These tuples are collected in a set $K$. The list to return is then obtained by extracting the keys of those tuples from $K$ in which the value is not $\perp$, i.e., those that have not been removed.

Note that the size of $D[i]$ could be reduced without affecting the emulation: operations that have been superseded by other operations (for the same key but with a larger timestamp) can be eliminated to save space.

We now show that this implementation produces linearizable histories that satisfy the specification of a KVS. Given a history $\sigma$, we construct a legal sequential history $\pi$ that satisfies the properties of linearizability and argue that it is legal.

- The empty history is legal.

- Any history without concurrent operations is legal. This follows because the timestamps are strictly monotonically increasing during all *put* and *remove* operations. As the *get* operation returns the value with the highest timestamp in tuple order, *get* returns the most recently written value. The same argument shows that the output of the *list* operation is legal.

- Two concurrent *get* and/or *list* operations with no concurrent *put* or *remove* operations can be ordered in any way, since they do not affect each other.

- Two concurrent *put* and/or *remove* operations with the same key are scheduled according to the order on the tuples that represent them. Suppose clients $p$ and $r$ are concurrently invoking operations $\omega_p = \mathit{put}$ and $\omega_r = \mathit{remove}$ with the same key. The operations are scheduled such that $\omega_p >_\pi \omega_r$ if and only if the tuple representing $\omega_p$ is bigger than the tuple representing $\omega_r$. Note that $p \neq r$ because all clients execute operations in a well-formed way. Thus, one of the two tuples is strictly bigger than the other in tuple order. All subsequent *get* and *list* operations also return the result of the operation ($\omega_p$ or $\omega_r$) that is scheduled last. Another subsequent put and remove operation with the same key will increment the timestamp, hence, the results of $\omega_p$ or $\omega_r$ are never returned afterwards.

- Consider now a *put* or *remove* operation that is concurrent to *get* or *list* operation. We observe that the *update* on $SO$ near the end of the *put* and *remove* operations and the *scan* on $SO$ at the beginning of the *get* and *list* operations are linearizable because $SO$ is atomic.

  We schedule a *put* or *remove* operation $\omega$ in $\pi$ before a concurrent *get* or *list* operation $\rho$ whenever the *update* in $\omega$ precedes the *scan* in $\rho$. If $\omega <_\pi \rho$, then $\omega$ incremented the timestamp and $\rho$ returns the value written by $\omega$ because it has the maximal timestamp, as required. Otherwise, if $\rho <_\pi \omega$, then $\rho$ appears in $\pi$ before $\omega$. This is also legal since the *scan* operation of $\rho$ has not been affected by the *update* operation in $\omega$ according to the construction of $\pi$. Hence, the value returned by $\rho$ is legal.

- Finally, consider multiple concurrent *put* and/or *remove* operations. As shown before, executions with one *put* or *remove* operation concurrent to one further operation can be linearized.

  Assume that $k - 1$ *put* and *remove* operations have been linearized and consider the $k$-th *put* or *remove* operation $\omega$. It is scheduled:

  - after all *get* and *list* operations whose embedded *scan* precedes the *update* operation in $\omega$;
  - before all *get* and *list* operations whose embedded *scan* is scheduled after the *update* operation in $\omega$;
  - before all *put* and *remove* operations by clients with a higher index; and
  - after all *put* and *remove* operations by clients with a smaller index.

  It is straightforward to verify that $\pi$ constructed like this is sequential and legal.

- History $\pi$ preserves the real-time order of $\sigma$ because no sequential operations are reordered.

This completes the construction of our wait-free implementation of a KVS object from a single-writer atomic snapshot object. Since the snapshot object has consensus number one, Theorem 1 implies that the consensus number of a KVS object is also one. □

**Algorithm 3** Implementation of a KVS object $K$ form a snapshot object $SO$.

**operation** *put*(*key*, *val*) by client with index $i$
    $D \leftarrow SO.scan()$;
    $t \leftarrow 0$;
    **for** $j \in \{1, \ldots, n\}$ **and** $(j, ts, k, v) \in D[j]$ **do**
        **if** $k = key$ **and** $t < ts$ **then**
            $t \leftarrow ts$;
    $t \leftarrow t + 1$;
    $d \leftarrow D[i] \parallel (i, t, key, val)$;
    $SO.update(i, d)$;
    **return**;

**operation** *get*(*key*) by client with index $i$
    $D \leftarrow SO.scan()$;
    $(t, val) \leftarrow (0, \perp)$;
    **for** $j \in \{1, \ldots, n\}$ **and** $(j, ts, k, v) \in D[j]$ **do**
        **if** $k = key$ **and** $t < ts$ **then**
            $(t, val) \leftarrow (ts, v)$;
    **return** *val*;

**operation** *remove*(*key*) by client with index $i$
    *put*(*key*, $\perp$);    // invoke the operation on itself
    **return**;

**operation** *list*()
    $D \leftarrow SO.scan()$;
    let $L$ be the set of distinct keys from $D$, i.e.,
        $L \leftarrow \left\{k \middle| (j, ts, k, v) \in \bigcup_{i=1}^{n} D[i]\right\}$;
    $K \leftarrow \emptyset$;
    **for** *key* $\in L$ **do**
        let $(j, ts, k, v)$ be the biggest tuple in $\bigcup_{i=1}^{n} D[i]$
            with $k = key$ according to tuple order;
        **if** $v \neq \perp$ **then**
            $K \leftarrow K \cup \{key\}$;
    **return** $K$;

# Appendix C

# Correctness of MOD-SMART

In this Appendix we prove the correctness of MOD-SMART, described in Chapter 6. The first theorem proves the safety of the protocol, i.e., that all correct replicas process the same sequence of operations.

**Theorem 22.** *Let $p$ be the correct replica that executed the highest number of operations up to a certain instant. If $p$ executed the sequence of operations $o_1, ..., o_i$, then all other correct replicas executed the same sequence of operations or a prefix of it.*

*Proof.* Assume that $r$ and $r'$ are two distinct correct replicas and $o$ and $o'$ are two distinct operations issued by correct client(s). Assume also that $b$ and $b'$ are the batches of operations were $o$ and $o'$ were proposed, respectively. For $r$ and $r'$ to be able to execute different sequences of operations that are not prefix-related, at least one of three scenarios described below needs to happen.

(1) *VP-Consensus instance $i$ decides $b$ in replica $r$, and decides $b'$ in $r'$.* Since in this scenario the same sequence number can be assigned to 2 different batches, this will cause $o$ and $o'$ to be executed in different order by $r$ and $r'$. But by the *Agreement* and *Integrity* properties of VP-Consensus, such behavior is impossible; *Agreement* forbids two correct processes to decide differently, and *Integrity* prevents any correct process from deciding more than once.

(2) *$b$ is a batch decided at VP-Consensus instance $i$ in both $r$ and $r'$, but the operations in $b$ are executed in different orders at $r$ and $r'$.* This behavior can never happen because Algorithm 11 (line 13) forces the operations to be ordered deterministically for execution, making these operations be executed in the same order by these replicas.

(3) *Replica $r$ executes sequence of operations $S = o_0, ..., o_s$ and $r'$ executes a subset of operations in $S$ (but not all of them), preserving their relative order.* This will result in a gap in the sequence of operations executed by $r'$. From Algorithm 11, we can see that any operation is executed only after the *VP-Decide* event is triggered. This event is triggered either when a consensus instance decides a batch in line 7 — which occurs during the normal phase — or when invoked by Algorithm 12 in line 41. For simplicity, let us assume that each batch of messages contains a single operation. In the absence of a synchronization phase, lines 3-6 of Algorithm 11 ensure that any consensus instance $i$ is only started after instance $i-1$ is decided. This forces any correct process to execute the same sequence of operations.

Lets now reason about the occurrence of a synchronization phase. In such case, $r'$ will create the *Log* set at Algorithm 12, and then trigger the *Decide* event for each decision contained in *Log* (lines 36-41). *Log* is created using operations from both the most up-to-date log contained in the SYNC message or from the replica's *DecLog* (line 36). Let us assume that $r'$ did not execute $S$ before entering the synchronization phase. Let us further consider $T = \{o_{s+1}, ..., o_t\}$ with $t \geq s+1$ to be a sub-sequence of operations that have been executed by $r$. For $r'$ to skip $S$ in this situation, it is necessary that *Log* contains $U$ (such that $U$ is a prefix of $T$) but does

not contain $S$, and that $r'$ triggers *VP-Decide* at Algorithm 12 (line 41) for each operation in $U$. This situation can never happen since $r'$ is correct, and the algorithm ensures *Log* is constructed using valid operations (satisfying `validDec`) from decision logs that contain no gaps, i.e., satisfy the `noGaps` predicate. Furthermore, each decision in *Log* also satisfied the `validDec` predicate, so $r'$ will not pick a sequence of operations with invalid decisions. Finally, since $r'$ is correct, $DecLog$ will already satisfy these predicates. This means that either: (1) both $S$ and $U$ are in $Log$; (2) only $S$ is in $Log$, (3) neither sequence is in $Log$. Therefore, if $Log$ contains $U$, then it must also contain $S$, and both sequences will be executed in $r'$. □

Next lemmata prove several MOD-SMART properties. These lemmata use the following additional definitions. We say that an operation issued by a client $c$ *completes* when $c$ receives the same response for the operation from at least $f + 1$ different replicas. We also consider that an operation sent by a client is *valid* if it is correctly signed and if its sequence number is greater than the sequence number of the last operation sent by that client.

**Lemma 9.** *If a correct replica receives a valid operation $o$, eventually all correct replicas receive $o$.*

*Proof.* We have to consider four possibilities concerning the client behavior and the system synchrony.

(1) *Correct client and synchronous system.* In this case, the client will send its operation to all replicas, and all correct ones will receive the operation and store it in the $ToOrder$ set before a timeout occurs (Algorithm 11, line 1-2 plus procedure *RequestReceived*).

(2) *Faulty client and synchronous system.* Assume a faulty client sends a valid operation $o$ to at least one correct replica $r$. Such replica will initiate a timer $t$ and start a consensus instance $i$ (Algorithm 11, lines 1 and 2 plus procedure *RequestReceived*). However, not enough replicas (less than $n - f$) will initialize a consensus instance $i$. Because of this, the timeout for $t$ will eventually be triggered on the correct replicas that received it (Algorithm 12, line 1), and $o$ will be propagated to all other replicas (lines 2 and 3). From here, all correct ones will store the operation in the $ToOrder$ set (Algorithm 11, lines 18 and 19 plus procedure *RequestReceived*).

(3) *Correct client and asynchronous system.* In this case, a correct replica might receive an operation, but due to delays in the network, it will trigger its timeout before the client request reaches all other replicas. Such timeout may be triggered in a correct replica and the message will be forwarded to other replicas. Moreover, since the client is correct, the operation will eventually be delivered to all correct replicas and they will store it in their $ToOrder$ set.

(4) *Faulty client and asynchronous system.* This case is similar to 3), with the addition that the client may send the request to as few as one correct replica. But like it was explained in 2), the replica will send the operation to all other replicas upon the first timeout. This ensures that eventually the operation will be delivered to all correct replicas and each one will store it in the $ToOrder$ set.

Therefore, if a correct replica receives a valid operation $o$, them all correct replicas eventually receive $o$. □

**Lemma 10.** *If a synchronization phase for regency $g$ starts with a faulty leader $l$, then eventually synchronization phase for regency $g' > g$ starts with correct leader $l' \neq l$.*

*Proof.* Each synchronization phase uses a special replica called 'leader', that receives at least $n - f$ STOPDATA messages and sends a single SYNC message to all replicas in the system (Algorithm 12, lines 24-29). If such leader is faulty, it can deviate from the protocol during

this phase. However, its behavior is severely constrained since it can not create fake logs (such logs are signed by the replicas that sent them in the STOPDATA messages). Additionally, each entry in the log contains the proof associated with each value decided in a consensus instance, which in turn prevents the replicas from providing incorrect decision values. Because of this, the worst a faulty leader can do, is:

(1) *Not send the* SYNC *message to a correct replica*. In this case, the timers associated with the operations waiting to be ordered will eventually be triggered at all correct replicas - which will result in a new iteration of the synchronization phase.

(2) *Send two different* SYNC *messages to two different sets of replicas.* This situation can happen if the faulty leader waits for more than $n-f$ STOPDATA messages from replicas. The leader will then create sets of logs $L$ and $L'$, such that each set has exactly $n-f$ valid logs, and sends $L$ to a set of replicas $Q$, and $L'$ to another set of replicas $Q'$. In this scenario, $Q$ and $Q'$ may create different logs at line 36 of Algorithm 12, and resume normal phase at different consensus instances. But in order to ensure progress, at least $n-f$ replicas need to start the same consensus instance (because the consensus primitive needs these minimum amount of correct processes). Therefore, if the faulty leader does not send the same set of logs to a set $Q_{n-f}$ with at least $n-f$ replicas that will follow the protocol (be them either all correct or not), the primitive will not make progress. Hence, if the faulty leader wants to make progress, it has to send the same set of logs to at least $n-f$ replicas. Otherwise, timeouts will occur, and a new synchronization phase will take place.

Finally, in each synchronization phase a new leader is elected. The new leader may be faulty again, but in that case, the same constraints explained previously will also apply to such leader. Because of this, when the system reaches a period of synchrony, after at most $f$ regency changes, there is going to be a new leader that is correct, and progress will be ensured. □

**Lemma 11.** *If one correct replica $r$ starts consensus $i$, eventually $n-f$ replicas start $i$.*

*Proof.* We need to consider the behavior of the clients that issue the operations that are ordered by the consensus instance (correct or faulty), the replicas that start such instance (correct or faulty), and the state of the system (synchronous or asynchronous).

We can observe from Algorithm 11 that an instance is started after selecting a batch of operations from the $ToOrder$ set (lines 4-6). This set stores valid operations issued by clients. From Lemma 9, we know that a valid operation will eventually be received by all correct replicas, as long as at least one of those replicas receives it. Therefore, it is not necessary to consider faulty clients in this lemma.

From the protocol, it can be seen that a consensus instance is started during the normal phase (Algorithm 11, line 6). Following this, there are two possibilities:

(1) *$r$ decides a value for $i$ before a timeout is triggered.* For this scenario to happen, it is necessary that at least $n-f$ processes participated in the consensus instance without deviating from the protocol. Therefore, $n-f$ replicas had to start instance $i$.

(2) *A timeout is triggered before $r$ is able to decide a value for $i$*. This situation can happen either because the system is passing through a period of asynchrony, or because the current leader is faulty. Let us consider a consensus instance $j$ such that $j$ is the highest instance started by a correct replica, say $r'$. Let us now consider the following possibilities:

*2-a) $r$ started $i$ and $i < j$*. Remember that our algorithm executes a sequence of consensus instance, and no correct replica starts an instance without first deciding the previous one (Algorithm 11, lines 3-6). If $i < j$, $j$ had to be started after $i$ was decided in $r'$. But if $i$ was decided, at least $n-f$ processes participated in this consensus instance. Therefore, $n-f$ replicas had to start instance $i$.

*2-b) $r$ started $i$ and $i > j$.* This situation is impossible, because if $j$ is the highest instance started, and both $r$ and $r'$ are correct, $i$ cannot be higher than $j$.

*2-c) $r$ started $i$ and $i = j$.* In this case, the synchronization phase might be initialized before all correct replicas start $i$. Because only a single correct replica might have started $i$, the log which goes from instance 0 to instance $i - 1$ might not be present in the SYNC message (sent by Algorithm 12, lines 28-29), even if all replicas are correct (because the leader can only safely wait for $n - f$ correct STOPDATA messages). This means that an instance $h$ such that $h \leq i$ will be selected by all correct replicas upon the reception of the SYNC message from the leader.

If the system is asynchronous, multiple synchronization phases might occur, where in each one a new leader will be elected. In each iteration, a faulty replica may be elected as leader; but from Lemma 10, we know that a faulty leader cannot prevent progress. Therefore, when the system finally becomes synchronous, eventually a correct leader will be elected, and $h$ will eventually be started by $n - f$ replicas.

Finally, let us consider the case where $h < i$. In this case, a total of $n - f$ replicas may start $h$ instead of $i$. But by the *Termination* property of our primitive, $h$ will eventually decide, and all correct replicas will start the next instance. Because of this, eventually $n - f$ replicas will start $i$, even if more synchronization phases take place. □

Using the previous lemmata we can prove that MOD-SMART satisfies the SMR Liveness with the following theorem.

**Theorem 23.** *A valid operation requested by a client eventually completes.*

*Proof.* Let $o$ be a valid operation which is sent by a client, and $I$ the finite set of consensus instance where $o$ is proposed. Due to Lemma 9, we know that $o$ will eventually be received by all correct replicas, and at least one of them will propose $o$ in at least one instance of $I$ (the *fair* predicate ensures this). By Lemma 11, we also know that such instances will eventually start in $n - f$ replicas.

Furthermore, let us show that there must be a consensus instance $i \in I$ where $o$ will be part of the batch that is decided in $i$. As already proven in Lemma 11, all correct replicas will eventually receive $o$. Second, we use the *fair* predicate to avoid starvation, which means that any operation that is yet to be ordered, will be proposed. Because of this, all correct replicas will eventually include $o$ in a batch of operations for the same consensus instance $i$. Furthermore, the $\gamma$ predicate used in the VP-Consensus ensures that (1) the operations in the batch sent by the consensus leader is not empty; (2) it is correctly signed; and (3) the sequence number of each operation is the next sequence number expected from the client that requested it.

Since there are enough replicas starting $i$ (due to Lemma 11), the *Termination* property of consensus will hold, and the consensus instance will eventually decide a batch containing $o$ in at least $n-f$ replicas. Because out of this set of replicas there must be $f+1$ correct ones, $o$ will be correctly ordered and executed in such replicas. Finally, these same replicas will send a REPLY message to the client (line 17, Algorithm 11), notifying it that the operation $o$ was ordered and executed. Therefore, a valid operation requested by a client eventually completes. □