D2.2.1
Preliminary Architecture 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 / D2.2.1 / 1.0
Activity and Work package contributing to deliverable:	Activity 2 / WP 2.2
Due date:	September 2011 – M12
Actual submission date:	3rd October, 2011

Responsible organisation:	FFCUL
Editor:	Marcelo Pasin
Dissemination level:	Public
Revision:	1.0 (r5726)

Abstract:	This document contains a presentation of the TClouds architecture, in its initial form. Besides the architecture, it describes a number of components for adaptive resilience, to be developed in the project.
Keywords:	Architecture, resilience, cloud storage, consistency, replication, Byzantine fault-tolerance.

Editor

Marcelo Pasin (FFCUL)

Contributors

Alysson Bessani (FFCUL)

João Sousa (FFCUL)

Marcelo Pasin (FFCUL)

Miguel Correia (FFCUL)

Paulo Veríssimo (FFCUL)

Pedro Costa (FFCUL)

Christian Cachin (IBM)

Johannes Behl (FAU)

Klaus Stengel (FAU)

Rüdiger Kapitza (FAU)

Davide Vernizzi (POL)

Emanuele Cesena (POL)

Gianluca Ramunno (POL)

Paolo Smiraglia (POL)

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 give a preliminary overview of the TClouds architecture, which is devoted to providing incrementally high levels of security and dependability to cloud infrastructures, in an open, modular and versatile way. To put matters in context, we start by discussing the motivation and requirements that make up the rationale for the model and architecture of TClouds, and we review the state-of-the-art in the area. Then we present the architecture specification, by introducing the main building blocks and enabling components. Then, we present several possible instantiations of the architecture addressing the security and dependability problems of as many realistic cloud-related scenarios. We conclude the report by introducing and discussing several components for adaptive resilience, to be designed in the project. These components are just a subset of the possible components that can be envisaged for TClouds, but they constitute a representative set allowing the development of several proof-of-concept prototypes.

Contents

1 Introduction **1**

2 Motivation and Requirements **3**

3 State of the Art and Basic Concepts **6**
3.1 Replication for Byzantine Failures 6
3.1.1 System Models 7
3.1.2 Replication Models 8
3.1.3 Byzantine Fault Tolerance 10
3.1.4 Extensions to the Programming Model 13
3.1.5 Conclusions 14
3.2 Cloud Storage 15
3.3 Computing in Clouds 16

4 Initial Architecture Specification **18**
4.1 Initial Architecture Specification 18
4.1.1 Overview of the key architecture aspects 18
4.1.2 Main Building Blocks 20
4.2 TClouds deployment alternatives 22
4.3 Adaptive Resilience 25
4.3.1 TClouds local node architecture 25
4.3.2 Incremental node resilience strategies 27
4.3.3 Making TIS resilient 27
4.3.4 TClouds distributed middleware devices 27

5 Components for Adaptive Resilience **32**
5.1 Object Storage 32
5.2 Consistency for untrusted service execution 32
5.3 State Machine Replication 33
5.4 Fault-tolerant workflow execution 35
5.5 BFT MapReduce 36
5.6 Logging 36

6 Object Storage **38**
6.1 Introduction 38
6.2 Cloud Storage Applications 39
6.3 The DEPSKY System 40
6.3.1 DEPSKY Architecture 40
6.3.2 Data Model 41
6.3.3 System Model 42
6.3.4 Protocol Design Rationale 43

6.3.5 DEPSKY-A– Available DepSky 44
6.3.6 DEPSKY-CA– Confidential & Available DepSky 45
6.3.7 Read Optimization 46
6.3.8 Supporting Multiple Writers – Locks 47
6.3.9 Additional Protocols 48
6.3.10 Dealing with Weakly Consistent Clouds 49
6.4 DEPSKY Implementation 50
6.5 Related Work 50
6.6 Conclusion 51

7 Consistency for untrusted service execution **52**
7.1 Introduction 52
7.1.1 Contributions 53
7.1.2 Organization 53
7.2 System model 53
7.2.1 System 53
7.2.2 Functionality 54
7.2.3 Operations and histories 54
7.2.4 Consistency conditions 54
7.2.5 Cryptographic primitives 55
7.3 Service execution and authentication 56
7.3.1 Separated execution 56
7.3.2 Authenticated separated execution 57
7.3.3 Examples 59
7.4 Fork-linearizable execution protocol 59
7.5 Conclusion 60

8 State Machine Replication **63**
8.1 Introduction 63
8.2 BFT-SMART Design Principles 63
8.3 The Replication Protocol 64
8.4 BFT-SMART Replica Architecture 66
8.4.1 Building blocks 66
8.4.2 Staged Message Processing 68
8.5 Using the BFT-SMART replication library 69
8.6 Performance 70
8.7 Lessons Learned 72
8.7.1 Making Java a BFT programming language 72
8.7.2 How to test BFT systems? 72
8.7.3 Dealing with heavy loads 73
8.7.4 Signatures vs. MAC vectors 73
8.8 Concluding Remarks 73

9 Fault-tolerant workflow execution **75**
9.1 Introduction 75
9.2 Basics About BPEL 76
9.3 Reliable BPEL Infrastructure 76
9.3.1 Overall Architecture 78

9.3.2 Implementation Details 78
9.4 Evaluation 79
9.5 Conclusion 81

10 BFT MapReduce **82**
10.1 Introduction 82
10.2 Hadoop MapReduce 83
10.3 BFT Hadoop MapReduce 83
10.4 Conclusion and Future Work 84

11 Logging **85**
11.1 Introduction 85
11.2 Log Service for a single cloud 85
11.3 Multi-cloud scenarios for the Log Service 86
11.3.1 Log as a Service (LaaS) 86
11.3.2 Cloud-of-Cloud-enabled Log Service 88
11.3.3 Cloud-of-Cloud-optimized Log Service 89
11.3.4 Cloud-of-Cloud-distributed Log Service 90
11.4 Concluding Remarks 90

Bibliography **90**

List of Figures

2.1 TClouds resilient cloud-of-clouds infrastructure: (a) Using untrusted cloud services; (b) Achieving cloud resilience with TClouds 4

4.1 TClouds middleware: (a) Services; (b) Use of services 19
4.2 TClouds architecture: (a) Block diagram; (b) TIS - TClouds Information Switch 21
4.3 TClouds-enabled client-resident software 23
4.4 TClouds-enabled cloud services mediator: client-side 23
4.5 TClouds-enabled cloud services mediator: server-side 24
4.6 Cloud provider solutions: (a) TClouds-enabled cloud provider; (b) Native TClouds cloud services provider 24
4.7 Diverse TClouds ecosystem 25
4.8 Local architecture of a TClouds node 26
4.9 Byzantine-resilient protocols 28
4.10 TClouds Appliance box implementation 29
4.11 Implementing a private cloud with a TClouds appliance box 29
4.12 Server-set implementation 30
4.13 Commodity provider evolution: (a) TClouds-enabled cloud; (b) TClouds native Data Center and Cloud 31

5.1 TCLODS object storage mapped onto its architecture. 33
5.2 State-machine replication mapped onto the TClouds architecture. 34

6.1 Architecture of DEPSKY (w/ 4 clouds, 2 clients). 41
6.2 DEPSKY data unit and the 3 abstraction levels. 42

8.1 The message pattern of a BFT-SMART fault-free execution. 65
8.2 The staged message processing on BFT-SMART. 68
8.3 API for using the BFT-SMART library at client and replica sides. 70
8.4 Latency vs throughput of BFT-SMART with messages of 4 bytes and 1 Kbyte when $f = 1$ and $f = 2$. 71

9.1 Standard unreplicated BPEL infrastructure. 77
9.2 Proposed architecture: BPEL engine and web services are actively replicated using output and input proxies (OP and IP). An automatic transformation of process definitions allows them to intercept the communication chain transparently. The proxies make use of a ZooKeeper service for coordination and dynamic configuration. 77

11.1 High level architecture of the Log Service. 86
11.2 *Log as a Service* scenario. 87
11.3 *Cloud-of-Cloud-enabled Log Service* scenario. 88
11.4 *Cloud-of-Cloud-optimized Log Service* scenario. 89
11.5 *Cloud-of-Cloud-distributed Log Service* scenario. 90

List of Tables

3.1 Comparison of BFT algorithms, expanding Table 1 in [KAD+07]. The throughput and latency metrics are for each request. f is the maximum number of faulty servers and b the size of the batch of requests used. (†) MinZyzzyva does 2 HMAC operations and one signature. 13

6.1 Functions used in the DEPSKY-A protocols. 45
6.2 Functions used in the DEPSKY-CA protocols. 46

8.1 Latency and peak throughput of SMaRt, PBFT and Spinning for requests with 4 bytes and $f = 1$. 71

9.1 Architecture comparison: Unreplicated with and without logging-based fault tolerance vs. transparent active replication via ZooKeeper and JGroups (times are in ms). 79

Chapter 1

Introduction

Chapter Authors:
Paulo Verissimo (FFCUL) and Marcelo Pasin (FFCUL)

Cloud Computing (CC) is a process and business model building on some recent technologies and paradigms, such as: Web services, storage as a service, inexpensive storage, service oriented architecture, on demand computing, grid computing, utility computing, virtualization, etc. CC seems to have definitely emerged as a model capable of organizing this forest of technologies and paradigms into a solid way of providing ubiquitous scale computing services (quoting NIST [JG11]): "... on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction".

Alongside with the advantages brought by CC, some challenges loom, and recent surveys have placed security and resilience as prime sources of concern, as companies and organizations seriously consider "cloudifying" their IT. As more and more services migrate to CC, so increases the dependence of the IT business on the latter. However, short of promising adequate security management of the infrastructure and perhaps some form of disaster recovery, there is little evidence of what 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 [Sar09, Nao09]. This scenario is bound to evolve positively under stakeholders pressure, at the very least by provision of (potentially proprietary) accredited cloud environments, but we believe that built-in, open, and diverse solutions to cloud dependability and security are required.

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

In this report we give a preliminary overview of the TClouds architecture and components, which are devoted to provide incremental levels of security and dependability to cloud infrastructures, in an open, modular and versatile way. To put matters in context, we start by discussing the motivation and requirements in Chapter 2, that make up the rationale for the model and architecture of TClouds, the we review the state-of-the-art in the area in Chapter 3. Then, in Chapter 4, we discuss how TClouds addresses several realistic resilient cloud computing scenarios, by presenting as many deployment alternatives. This objective will be attained by offering the designer different instantiations of the architecture, addressing the security and dependability problems put by each scenario. To keep complexity to a manageable level, the TClouds architecture should serve these objectives essentially by re-using and reconfiguring the same basic components providing trusted IaaS and PaaS services. In particular, we discuss implementations of TClouds functionality from minimal changes preserving the use of legacy commodity clouds IaaS, to more ambitious steps, such as commodity cloud provider migration to native TClouds.

Chapter 5 follows the architecture presentation by introducing and discussing several components for adaptive resilience, to be designed in the project. These components are just a subset of the possible components that can be envisaged for TClouds, but they constitute a representative set allowing the development of several prototypes as a proof-of-concept of the architecture's effectiveness in promoting open, modular and versatile resilient cloud computing.

Specific descriptions of the cloud-of-clouds components under design and development in TClouds (called subsystems in WP2.4) appear in Chapters 6 to 11. Chapter 6 describes an Object Storage subsystem built on top of commodity cloud storage. Chapter 7 presents solutions for ensuring consistency when running services in untrusted clouds. State Machine Replication is proposed in Chapter 8 for executing trusted services under Byzantine failures. Chapter 9 proposes a fault-tolerant component to execute workflows. Chapter 10 presents a modified Mapreduce platform-as-a-service, in order to tolerate Byzantine failures. The report concludes with Chapter 11, proposing cloud-of-clouds logging services.

Chapter 2

Motivation and Requirements

Chapter Author:
Paulo Verissimo (FFCUL)

In this section, we discuss the motivation and requirements that make up the rationale for 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 described succintly by the following:

- A "cloudified" 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.
- However, specific and proprietary IaaS or PaaS approaches to achieving security can make migration or interoperation difficult and expensive, creating vendor lock-in and competition exclusion.
- Finally, even open approaches to these problems, if confined to a single-cloud provider, will not address high-resilience objectives, since they are, at organizational level, a single point of failure.

These problems can however be solved and we propose to address them in 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, as depicted in Figure 2.1b. This infrastructure should in essence provide automated computing resilience against attacks and accidents in complement or in addition to commodity clouds.

The functionality of this infrastructure will be defined ahead, and in order to serve a set of requirements of resilient cloud computing. We state below the necessary requirements, by lining up a set of propositions which translate into desirable macroscopic properties of the system, and by discussing their rationale. Consequently, the architecture will be developed having in mind the requirements imposed by these propositions. This way, the reader and/or potential developer or user can get a clear view of what is behind the architectural options proposed for TClouds and, vice-versa, can gain confidence that the architecture and respective algorithms and middleware are bound to satisfy the imposed requirements.

Complement classical security techniques with resilience mechanisms

Classical security techniques are largely based on prevention, human intervention and ultimately disconnection. There is thus a need for achieving tolerance, automation and availability,

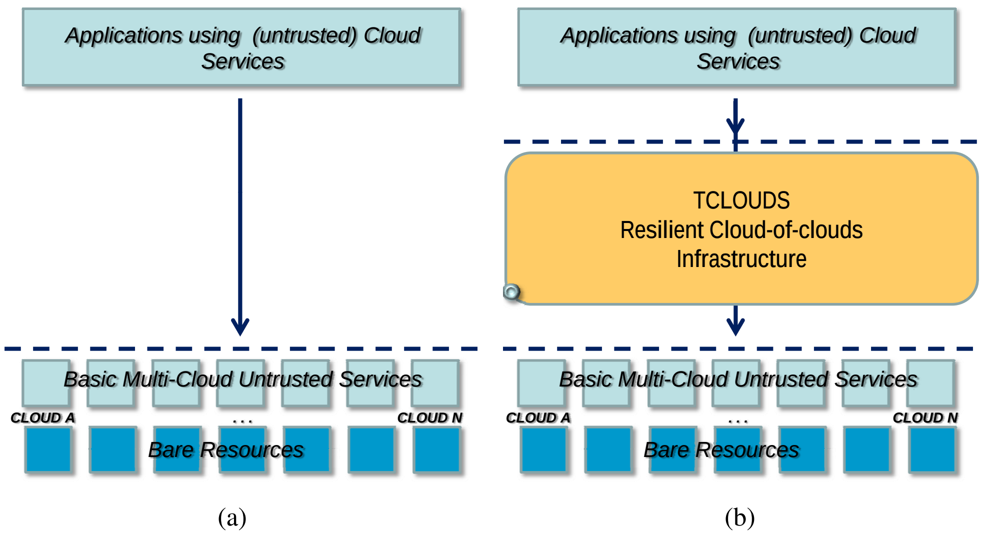

Applications using (untrusted) Cloud Services
Basic Multi-Cloud Untrusted Services
CLOUD A
...
CLOUD N
Bare Resources
(a)
Applications using (untrusted) Cloud Services
TCLOUDS
Resilient Cloud-of-clouds
Infrastructure
Basic Multi-Cloud Untrusted Services
CLOUD A
...
CLOUD N
Bare Resources
(b)

Figure 2.1: TClouds resilient cloud-of-clouds infrastructure: (a) Using untrusted cloud services; (b) Achieving cloud resilience with TClouds

both under attack and in the presence of major accidents [VNC+06].

Promote automatic control of macroscopic information flows

In such complex, large-scale and distributed infrastructures, any solution, to be effective, has to involve automatic mechanisms to secure the macroscopic command and information flows between the major modules, such as: between layers of different trustworthiness, from unprotected commodity cloud layers up to the end user; amongst peer layers implementing resilience-improving mechanisms [VNC08].

Preserve legacy needs whilst enabling a diverse ecosystem

One 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.

Avoid single points-of-failure

This objective is at the very least meaningful at individual cloud level, by foreseeing mechanisms providing availability, privacy and integrity at overall service level. However, it is also very relevant in the sense of foreseeing mechanisms avoiding dependence on a single cloud provider, which at organizational level is also a single point-of-failure. This points to exploiting the redundancy and diversity that comes from relying on multiple cloud providers (clouds-of-clouds). The relevant mechanisms should, however, be as transparent as possible to users.

Address multiple resilient cloud computing deployment alternatives

An architecture having in mind addressing some of the objectives above should also take into account the multiple facets of the cloud computing business, offering multiple deployment alternatives for resilience. However, to avoid an explosive growth of complexity, this 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 cloudification" (allowing an organisation to build its own resilient private cloud); support of lightweight end-users directly over commodity clouds.

Chapter 3

State of the Art and Basic Concepts

Chapter Authors:
Alysson Bessani (FFCUL), Marcelo Pasin (FFCUL), Miguel Correia (FFCUL), Paulo Veríssimo (FFCUL), Christian Cachin (IBM), Johannes Behl (FAU), Klaus Stengel (FAU), Rüdiger Kapitza (FAU) and Davide Vernizzi (POL)

3.1 Replication for Byzantine Failures

Today's information society increasingly depends on computer-provided services. This becomes evident the minute these services are no longer accessible due to faults and, even worse, in the catastrophic case when faulty results are provided to users. While the first category can be treated by standard replication solutions, the second category, so-called Byzantine failures, presenting themselves in the form of software bugs, intrusions, viruses and hardware errors, is far more difficult and resource intensive to handle. So far, industry has been reluctant to employ approaches to address Byzantine failures generically as associated financial costs are considered as too high. However, there is an ongoing trend in our society to rely more and more on IT-based solutions and recent studies indicate that, for instance, non-benign hardware errors are more frequent than previously assumed. Furthermore, rate as well as severity of bug reports concerning all kinds of software appliances, is, despite numerous countermeasures, still high. Accordingly, one can draw two conclusions: best practices to avoid software as well as hardware errors are not enough and due to the rising importance of IT-based systems, new ways to provide dependable systems need to be explored. As a consequence, this chapter details approaches to tolerate Byzantine faults as they are a promising technology to make future IT solutions more robust and dependable. We consider cloud computing and especially intra-could replication as an ideal starting point to initiate this development. As of today, cloud and large cluster systems already employ replication to ensure the availability of important information necessary for configuring and coordinating large distributed applications. Prominent examples are ZooKeeper developed by Yahoo and the Chubby lock service. So far, these services only tolerate benign faults and a typical installation comprises five service instances, which allows to tolerate up to two simultaneous failures. Making these services Byzantine fault tolerant is very attractive and necessary, since they might be directly exposed to the Internet and are central for distributed applications that potentially control a large number of resources.

In the course of this chapter, we will give a brief introduction into BFT and present a set of initial systems showing that BFT can be practical and provides sufficient performance to legitimate further research for building the basis of intra-cloud middleware. While some of these systems already demonstrated their usability by replicating real life services (e.g., a network file system (NFS)) these systems still suffer from multiple problems such as resource demand, limited or even no support for parallel execution as well as missing support for diversity. The first aspect can be directly translated to high working costs, the second strongly constraints throughput and the last aspect basically decreases the degree of fault tolerance.

In fact, support for parallelism as well as support for diversity is largely orthogonal to the actual replication infrastructure that, at its core, executes a distributed protocol. However, as both lack a sufficient and generic solution, they can be seen as inhibitors for the wide spread use of BFT. Fortunately, recent research efforts in the field of deterministic execution indicate that this is addressed now and solutions are readily available. Furthermore, we consider diversity as a less pressing issue as there is a common interest to make standalone systems heterogeneous (e.g., at the process level using address space randomization) to reduce the success of viruses and similar malware. This leaves resource demand as a core problem. Traditionally, BFT demands for 3f+1 replicas to tolerate f Byzantine faults, meaning in a minimal setting 4 service instances to tolerate a single Byzantine fault. To circumvent this theoretical barrier, one can apply a hybrid fault model that basically assumes that some limited part of a system can only fail by crashing. This way, the resource demand of a BFT system can be reduced to the demand of a classical crash stop system which requires only 2f+1 replicas. Accordingly, we discuss recent results following this direction of research.

In the following section, we will first detail system requirements for BFT. Next, Quorum as well as state machine replication will be discussed. In doing so, practical aspects as well as recent advances to reduce the resource demand of BFT will be detailed. In the end of the section, aspects concerning the programming model like diversity and parallel request processing will be investigated and limitation of current approaches are identified.

The chapter is concluded with two sections: one on cloud storage, another on cloud processing. The first of them presents current solutions to support high-availability and trustworthiness from cloud storage, showing what could be done to improve them. The second and last section describes the problems of ensuring the integrity of operational data and outsourced computations.

3.1.1 System Models

All works described in this chapter consider a distributed system composed by a set of processes (clients and servers) interconnected by point-to-point channels.

A fundamental notion of a system model is its synchrony which defines how strong timing assumptions of the model are. At one end of the spectrum of all possible timing models, one can find the *asynchronous distributed system model*, in which time is regarded as completely absent [FLP85]. At the other end, there is the *synchronous system model*, in which every communication and computation takes at most a bounded and known amount of time. Between the many intermediate models of synchrony, one is especially relevant for BFT replication protocols: the *eventually synchronous system model* [DLS88]. In this model, there is a bound Δ and an instant GST (Global Stabilization Time) for all executions of the system, so that every message sent by a correct process to another correct process at instant $u >$ GST is received before $u+\Delta$, with Δ and GST unknown. The intuition behind this model is that the system can work asynchronously (with no bounds on delays) most of the time, but there are stable periods in which the communication and processing delays are bounded[1]. This model is particularly important, because it

[1]In practice, this stable period has to be long enough for the distributed computation to terminate, but does not need to hold forever.

represents the best effort networks like the Internet or the ones employed inside a datacenter. Unless stated otherwise, all protocols described in this chapter follow this model.

All components of the systems we consider can be subject to faults. Usually, it is said that a distributed system composed by n processes can tolerate at most $f < n$ faults. The bound f is called fault-threshold. The three most popular fault models are crash, crash-recovery and Byzantine. In the crash fault model, also called fault-stop, a faulty process just halts and stops interacting with other components of the system for the remaining of the execution. The crash-recovery model is a variation of the crash fault model in which faulty processes may recover after some time. In the Byzantine or arbitrary fault model [LSP82a], a faulty process may deviate arbitrarily from its specification and can exhibit any behavior. This last fault model is in particular relevant for the TClouds project since a faulty process in the Byzantine model (also called a Byzantine process) can be used to model a process controlled by a malicious adversary, that is a process having been subject of an intrusion [VNC03]. Due to this reason, unless said otherwise, we always consider protocols and systems tolerant to Byzantine faults, usually called BFT systems.

3.1.2 Replication Models

One key mechanism for implementing fault-tolerant services is replication. In a replicated system there is an unbounded set of clients interacting with a set of n servers acting as service replicas. Communication between clients and servers as well as among the servers is carried out according a protocol that implements a replication scheme.

Since most replication work in TClouds revolves around the tolerance of Byzantine faults, we preclude the use of the replication model usually employed for (crash only) fault tolerance: passive (or primary-backup) replication [BMST93]. In this model there is a primary replica that executes all operations issued by the clients, and forwards the results of these operations to a set of backup replicas able to take over the primary role in case of failures.

The problem in using passive replication with Byzantine failures is that a malicious primary may execute operations in a wrong way to fool both clients and backup replicas. Since the reply sent by the primary is not verified or compared with the result of the execution of the operation in other replicas (in fact, the backup replicas do not execute the request, only store the update result), there is no way to verify the correctness of the primary's computation.

Given this limitation, in the remaining of this section we discuss two classical replication models in which clients effectively compare results from different replicas to extract meaningful responses.

Quorum Systems

Quorum systems are a technique for implementing dependable shared memory objects in message passing distributed systems [Gif79]. Given a universe of data servers, a quorum system is a set of server sets, called *quorums*, that have a non-empty intersection. The intuition is that if, for instance, a shared data block is stored on all servers, any read or write operation has to be done only in a quorum of servers, not in all of them. The existence of intersections between the quorums allows the development of read and write protocols that maintain the integrity of the shared variable even if these operations are performed in different quorums.

Byzantine quorum systems are an extension of this technique for environments in which clients and servers can be subject of Byzantine failures [MR98a]. Formally, a Byzantine quorum system is a set of server quorums in which each pair of quorums intersect in sufficiently many servers (*consistency*) and there is always a quorum in which all servers are correct (*availability*). The servers can be used to implement one or more shared memory objects. Among the many types of quorum systems, two of them are fundamental for implementing Byzantine replication.

In the first type, the servers form a f*-masking quorum system* that tolerates at most f faulty servers, that is, it masks their failures [MR98a]. This type of Byzantine quorum systems requires the correctness of the majority of the servers in the intersection between any two quorums, i.e., $\forall Q_1, Q_2 \in \mathcal{Q}, |Q_1 \cap Q_2| \geq 2f+1$. Given this requirement, each quorum of the system must have $q = \lceil \frac{n+2f+1}{2} \rceil$ servers and the quorum system can be defined as: $\mathcal{Q} = \{Q \subseteq U : |Q| = q\}$. This implies in $|U| = n \geq 4f+1$ servers.

The second type is called f*-dissemination quorum system* in which a value is disseminated among n servers despite the existence of f faulty servers. This type of system requires data to be *self-verifiable*, i.e., any faulty server that corrupts its replica of the shared object will be detected. For such systems, at least one correct server is required in the intersection between any two quorums, thus $\forall Q_1, Q_2 \in \mathcal{Q}, |Q_1 \cap Q_2| \geq f+1$. Given this requirement, each quorum of the system must have $q = \lceil \frac{n+f+1}{2} \rceil$ servers and the quorum system can be defined as: $\mathcal{Q} = \{Q \subseteq U : |Q| = q\}$, which implies $|U| = n \geq 3f+1$ servers.

With these constraints, a f-masking quorum system (resp. f-dissemination quorum system) with $n = 4f+1$ (resp. $n = 3f+1$) will have quorums of $3f+1$ servers (resp. $2f+1$ servers).

State Machine Replication

A natural way to make a service fault-tolerant is to model it as a deterministic state machine, and replicate the service implementation over a set of servers, while ensuring that all of them start with the same state and execute the same sequence of operations. This is the main idea behind the *State Machine Replication* (SMR) model [Sch90], also called active replication.

In this model, the servers, hereafter called replicas, receive an operation request issued by a client, process it (possibly) modifying their states and send a reply. Formally, a state machine replication is characterized by three properties:

1. *Initial state* : All correct replicas start in the same state;
2. *Determinism* : Two correct replicas on identical states that execute the same request go to the same next state and generate equal results;
3. *Coordination* : All correct replicas receive and execute the same sequence of requests.

Although property 1 is trivial to implement, property 2 severely constraints the kind of services that one can replicate using this technique. The problem is that by requiring determinism, the state machine replication model rules out the possibility of replicas to independently generate timestamps, random numbers or even to run multiple threads, due to the fact of the inherent non-determinism of these actions that can lead to divergent states of correct replicas. Some work has been devoted to replicate non-deterministic state machines (e.g., [CRL03, KSC+10]), but they are still not supported by practical implementations.

Property 3 requires the use of a total order broadcast protocol. The idea is to make clients issue their requests through this communication primitive ensuring that all replicas receive (and process) the same sequence of requests.

Quorum Systems vs. State Machine Replication

One advantage of quorum systems in comparison to the state machine approach is that they do not require that operations are executed in the same order at all replicas, so they do not need to solve consensus. Quorum protocols usually scale much better due to the opportunity of concurrency in the execution of operations and the shifting of hard work from servers to client processes [AEMGG+05]. On the other hand, pure quorum protocols cannot be used to implement objects stronger than register (in asynchronous systems), on the contrary of state machine replication, which is more general [ES05] (but require additional assumptions).

3.1.3 Byzantine Fault Tolerance

After presenting the main assumptions required for implementing replication for critical services and discussing the two main replication models, we now describe the most relevant works on Byzantine fault-tolerant (BFT) replication in the context of the TClouds project.

Practical Byzantine Fault Tolerance

In this section we discuss the most important works showing that BFT replication can be efficient enough to be used in practice.

Castro-Liskov BFT. The CL-BFT is a state machine replication protocol in which a total order broadcast algorithm is used to ensure that all replicas execute all operations issued to them in the same order [CL02]. This total order broadcast protocol is based on a leader that sends sequence numbers for each operation issued to the replicated service. The system replicas execute two communication rounds of message exchanges in order to be sure that the order defined by the leader is correct. An important part of this protocol is the leader election algorithm, triggered when the leader do not send a sequence number for a message or send different sequence numbers to different replicas. CL-BFT always ensures linearizability (i.e., the replicated deterministic service emulates a corresponding non-replicated one [HW90a]) but liveness is satisfied only if the assumptions defined by the eventually synchronous model are satisfied. The basic CL-BFT replication protocol requires $3f+1$ replicas and five communication steps to execute an operation, however, some optimizations can be used to reduce the latency by one step [CL02, MA06].

Q/U. The work by Abd-El-Malek et al. aims to implement general services using quorum-based protocols in asynchronous BFT systems. Since this cannot be done ensuring unconditional termination (called wait-freedom [Her91] in the distributed computing parlance), the approach sacrifices liveness: the operations are only obstruction-free [HLM03], i.e., an operation is guaranteed to terminate only if there is no other operation executing concurrently. The main benefit of Q/U is its fault scalability: it attains high throughput even when the number of faults tolerated is high. Moreover, since only basic quorum-based protocols are employed, the Q/U protocols have linear message complexity and expected small latency (two communication steps). On the other hand, Q/U has mainly two drawbacks when compared with other BFT algorithms: *(i.)* it is, as stated before, only obstruction-free, so in a Byzantine-prone environment malicious clients could invoke operations continuously, causing a denial of service; and *(ii.)* it requires at least $5f+1$ servers instead of $3f+1$, which is the lower bound on the number of servers required for BFT, and it has a further impact on the costs of the system due to the costs of diversity [GPS07, GBG+11].

HQ-Replication. Cowling et al. proposed HQ-REPLICATION [CML+06], an interesting replication scheme that uses quorum protocols when there is no contention in an object access and consensus protocols to resolve contention situations. HG requires $n \geq 3f+1$ replicas and processes reads and writes in two (four when write-backs are needed) and four communication steps, respectively, in contention-free executions. When contention is detected (in reads too, due to the write-back), the system uses CL-BFT protocol [CL02] to order contending requests. This contention resolution protocol adds great latency, reaching more than ten communication steps even in synchronous and failure-free executions. It is true that in contention-free executions, HQ is expected to be very fast, however, in unpredictable networks (in terms of transmission delay) and Byzantine-prone environments, the usual assumption that contention will happen rarely is not obvious for two reasons: *(i.)* If the replicas are deployed in different domains (a required deployment strategy to enforce faulty independence [OBLC06]), this can lead to unexpected communication delay between clients and different replicas in situations with high loads; and *(ii.)* there is a possibility that Byzantine clients execute operations on the system only to create contention and consequently make it less efficient.

Zyzzyva. Kotla et al. [KAD+07] proposed a protocol that uses speculation to decrease the expected latency of BFT replication. The protocol is mainly based on a recent insight that shows that Paxos-based Byzantine consensus can be executed in two communication steps [MA06]. This insight is applied to make a speculative BFT replication protocol: The client receives the responses from the servers and knows if the operation result was correct, but the servers do not know if the result sent by them is in accordance with the other servers. Results and server states are called *speculative* and for ensuring the system correctness despite failures, previous state must be stored to be rolled back if some inconsistencies are found during the speculative execution. One of the key ideas of ZYZZYVA is that inconsistencies on system state are verified by clients that inform the servers. The resulting protocol is very efficient when there are no server failures and communications are timely. In this case, only three communication steps are required and linear message complexity is presented. However, there are two drawbacks in ZYZZYVA. First, a client must wait for equal replies from all $3f+1$ replicas to be able to use the speculative response (instead of the usual $2f+1$). This means that a timer must be set to wait for these replies (if there are faults, up to f servers could not send replies). A recent study presented in [JMM07] shows that some "fast" protocols perform worst than theoretically slower ones due to the requirement of waiting replies from more servers. This happens because in real networks, there is almost always some communication link connecting a client and a replica that requires much more time to deliver messages. Second, since ZYZZYVA uses speculative states, the servers must store several versions of the system state until the replicas state is committed. Besides the algorithmic complications due to the management of several speculative states of the system, the manipulations of these state versions can have significant processing and transmission costs, especially if there is a lot of object state processed by the system.

BFT under Non-favorable Conditions

Two papers describe the performance of some BFT protocols under non-optimistic assumptions. In [SDM+08] the performance of several BFT protocols under different configurations and network conditions is examined. This study shows that there is no protocol that is good in all possible conditions and that current optimistic protocols behavior can differ substantially depending of the execution environment properties. Although this work shows that there are some ignored problems with these protocols, it does not study the performance of them under attack. Amir et. al [ACKL08] identify possible malicious behaviors of faulty replicas using the CL-BFT protocol that can turn them very inefficient, and propose a performance-oriented criteria (prime-stability) and a new protocol that satisfies this criteria (PRIME) even in faulty conditions. Unfortunately, the prime protocol is much more costly than current optimistic protocols presented within the previous section due to the cost of the extra communication steps required by the protocol and the cryptographic mechanisms introduced to nullify the possibility of a malicious replica to delay its execution.

More recently, Clement et al. proposed the Aardvark algorithm that modifies PBFT in order to protect it from attacks against performance [CWA+09a]. The architecture defined in this work advocates the use of resource isolation to make correct replicas tolerate flooding attacks (DoS) from malicious replicas and clients. Moreover, Aardvark also changes the primary whenever the primary seems to be performing slowly, but it does this change by running a view change operation, limiting the damage done by a performance degradation attack launched by a faulty primary.

Spinning is another protocol in which the primary role keeps changing constantly in such a way that each replica is a primary for at most one consecutive ordering protocol execution [VCBL09]. This simple idea makes the protocol very efficient in absence of faults and dilutes the performance degradation imposed by faulty replicas.

Finally, there exists a set of completely decentralized protocols that do not rely on primary replicas to propose order for messages: the randomized protocols. The most efficient BFT replication library based on this technique is RITAS [MNCV06]. This system shows that randomization-based protocols, which have very interesting features such as absence of a leader and signature-freedom, can be efficient and present competitive performance values (latency and throughput) when compared with other leader/timing-based protocols.

The Use of Trusted Components for Improved BFT

Most works about BFT replication use a *homogeneous fault model*, in which all components can fail in the same way, although bounds on the number of faulty components are established (e.g., less than a third of the replicas). With this fault model and a non-synchronous system model it has been shown that it is not possible to do Byzantine fault-tolerant state machine replication with less than $3f+1$ replicas [Tou84].

The idea of using a hybrid fault model in the context of intrusion tolerance or Byzantine fault tolerance, was first explored in the MAFTIA project with the TTCB work [CLNV02]. The idea was to extend the replicas with a tamper-proof subsystem. It was in this context that the first $2f+1$ state machine replication solution appeared [CNV04]. However, it was based on a distributed trusted component, with a harder to enforce tamperproofness, specially in wide area networks.

More recently Chun et al. presented another $2f+1$ BFT algorithm based on similar ideas, A2M-PBFT-EA [CMSK07]. This algorithm requires only local tamper-proof components, dubbed Attested Append-Only Memory (A2M). The A2M is an abstraction of a trusted log. A2M offers methods to append values and to look up values within the log. It also provides a method to obtain the end of the log and to advance the suffix stored in memory (used to skip ahead by multiple sequence numbers). There are no methods to replace values that have already been assigned. The main goal of this trusted component is to provide a mechanism for algorithms to become immune to duplicity. Replicas using the A2M are forced to commit to a single, monotonically increasing sequence of operations. Since the sequence is externally verifiable, faulty replicas can not present different sequences to different replicas.

A latter paper [LDLM09] shows how the A2M abstraction can be implemented using only the TCG Trusted Platform Module (TPM) [Gro07a, Gro07b], a standard trusted computing platform chip present in many modern computers. The implementation is based on the use of monotonic counters, a TPM service that appeared only in version 1.2. The TCG specifications mandate the implementation of four monotonic counters in the TPM, but also that only one of them can be used between reboots [Gro07a]. Sarmenta et al. override this limitation by implementing virtual monotonic counters on an untrusted machine with a TPM [SvDO+06]. These counters are based on a hash-tree-based scheme and the single usable TPM monotonic counter. These virtual counters are shown to allow the implementation of count-limited objects, e.g., encrypted keys, arbitrary data, and other objects that can only be used when the associated counter is within a certain range.

Concurrently with this previous work, Veronese et. al. propose a new BFT protocol called MinBFT that also uses the TPM monotonic counters to build an abstraction called USIG (Unique Sequential Identifier Generator) [VCB+09]. In their work this abstraction was used to build state machine protocols that are optimal in required number of replicas ($2f+1$ instead of $3f+1$) and number of communication steps (4 instead of 5). The reduced number of communication steps is a fundamental difference between MinBFT and A2M-PBFT-EA implemented as in [LDLM09]. Besides MinBFT, Veronese et. al. also shows how to use the USIG service to implement a version of Zyzzyva requiring only $2f+1$ replicas called MinZyzzyva.

Table 8.1 shows a comparison of some of the most important protocols discussed in this section.

		PBFT [CL02] (+[YMV+03])	Zyzzyva [KAD+07]	TTCB [CNV04]	A2M-PBFT-EA [CMSK07]	MinBFT [VCB+09]
Model	Trusted Comp.	no	no	TTCB	A2M	USIG
Speculative		no	yes	no	no	no
Cost	Total replicas Replicas with state	$3f+1$ $2f+1$ [YMV+03]	$3f+1$ $2f+1$	$2f+1$ $2f+1$	$2f+1$ $2f+1$	$2f+1$ $2f+1$
Throughput	HMAC ops	$2+\frac{(8f+1)}{b}$	$2+\frac{3f}{b}$	3	$2+\frac{(2f+4)}{b}$	$2+\frac{(f+3)}{b}$
Latency	Num. comm. steps	5 / 4	3	5	5	4

Table 3.1: Comparison of BFT algorithms, expanding Table 1 in [KAD+07]. The throughput and latency metrics are for each request. f is the maximum number of faulty servers and b the size of the batch of requests used. (†) MinZyzzyva does 2 HMAC operations and one signature.

3.1.4 Extensions to the Programming Model

Layered BFT. Yin et al. presented a BFT algorithm for an architecture that separates agreement (made by $3f+1$ servers) from service execution (made by $2f+1$ servers) [YMV+03].

This was an important contribution to the area because service execution is expected to require much more computational resources than agreement. However, agreement still needs $3f+1$ machines, while in the present work we need only $2f+1$ replicas also for agreement.

An interesting extension of this model is proposed by Wood et. al. in a very recent paper [WSV⁺11]. The approach, named ZZ, exploits the use of VMs on the execution layer to implement services requiring only $f+1$ replicas in fault-free and synchronous executions (expected to be a common case in datacenters), starting more machines on-demand if divergence of replies is detected.

Parallel processing of requests. In traditional BFT systems [CL02], correct replicas process requests sequentially in the order determined by a total order broadcast protocol to ensure consistency. For multithreaded service implementations, this practice usually leads to a performance hit as application parallelism cannot be exploited. To mitigate this problem, Kotla et al. [KD04] proposed to relax the order in which requests are executed on different replicas by using application-specific knowledge to safely process *independent* requests in parallel; two requests are independent, if they both perform read operations, or if they access (read or modify) different parts of the application state.

ODRC [DK11] extends the idea of executing independent requests in parallel by dividing the overall application state machine into multiple object state machines, one for each state object. Using this approach, only requests accessing the same object have to be processed sequentially to guarantee replica consistency. ODRC also supports requests accessing multiple objects by synchronizing the object state machines involved on demand.

Storyboard [KSC⁺10] allows an even greater degree of parallelism by limiting sequential execution to critical sections only. In case two threads ask to acquire a lock protecting the same critical section, a customized thread library ensures that the threads enter the critical section in the order of the requests they execute.

Diversity. Common vulnerabilities and bugs that affect more than one replica of a replicated system can break the assumption that the system tolerates f faults since a single fault (a common vulnerability) can bring down more than one replica. To deal with this problem, the use of diverse replicas is a common assumption. There is a common understand that the use of N-version programming [AC77] is too costly for practical systems, however, opportunistic diversity of operating systems and database management systems were shown to be very effective to avoid common faults and bugs on these types of components [GPS07, GBG⁺11].

Even knowing that diversity is indeed effective in preventing common mode faults and vulnerabilities, there still remain the problem of integrating these diverse components in a single (replicated) system. The work described in [CRL03] advocates the idea of using abstraction layers to deal with this problem. It proposes to implement diverse versions of a BFT file system and a BFT object database above the CL-BFT protocol [CL02]. Gashi et. al. also present a prototype of a replicated system integrating diverse databases [GPS07]. Their approach revolves around using generic database access libraries like ODBC and JDBC to access these databases using the same code.

3.1.5 Conclusions

Using BFT protocols as a basis for intra-cloud middleware seems to be unavoidable when recent studies about hardware faults and software vulnerabilities are taken into account. This gets even more clear considering that cloud infrastructures need central services to coordinate and manage large resource pools that are publicity accessible. This makes these services to neuralgic spots of cloud infrastructures and to high value targets for attackers.

We identified three shortcomings to enable wide spread use of BFT: resources, parallel execution, and diversity. The resource demand problem can be solved by resorting to a hybrid fault model. The problem of parallel execution and the demand to enforce replica determinism is not specific to BFT, but a general problem or state machine replication. We presented some solutions that reduce the problem but demand for manual intervention. Despite the encouraging results about the effectiveness of diversity [GPS07, GBG+11], there are still areas that deserve more work. Diversity can be enable at different levels and at varying degree. At the momenti, there is no generic and widely applicable solution that is effective and cost efficient.

In large parts, the provided information can be projected to a service replication in a clouds-of-clouds scenario. Furthermore, such a scenario can be beneficial in terms of diversity as factors like natural disasters or bad administration are addressed. As an additional factor, latency has to be taken into account, however, there is already a fair number of proposals to considering that.

3.2 Cloud Storage

The increasing maturity of cloud computing technology is leading many organizations to migrate their computing infrastructure to the clouds. Some are fully embracing the cloud, others are adapting their software solutions to operate partially in the cloud. Even governments and companies that maintain critical infrastructures (as healthcare or telecommunications) 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 TClouds we intend to leverage the benefits of cloud computing by using a combination of diverse commercial clouds to build a trustworthy *cloud-of-clouds*.

Unavailability can be experienced when accessing data in the cloud. Such accesses rely on the Internet, which can be partially unavailable at times. Denial-of-service attacks can also cause unavailability, as happened with a service hosted in Amazon EC2 in 2009 [Met09]. Furthermore, a subsidiary of Microsoft lost contact names and photos of a large number of Sidekick users in 2009 [Sar09], while Ma.gnolia lost hundreds of megabytes of users data [Nao09]. TClouds components exploit Byzantine fault-tolerant replication and diversity on several clouds, allowing access to services as long as a subset of them is available and keeping the service correct even in cases of data loss or corruption.

Data stored in the cloud as well as access patterns are subject to the provider's discretion, especially in applications involving private data like health records or secret data as information about new products being developed by a company. Cryptography may be used, but the keys must be transferred to the cloud if data records have to be processed there, and malicious insiders are a problem even in diligent providers.

Many users are concerned to move to the clouds and be locked inside, as providers may become dominant [ALPW10], or it may cost too much to move all data out of it. This concern is amplified in Europe, as all major players are in the United States. TClouds protocols allow to store fragments of encrypted data at multiple cloud providers and to reduce costs associated with data access.

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 for single cloud offerings [MJWS10]. Work in TClouds 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 backup [VSV09], implement a database [BFG+08] or add provenance to the stored data [MRMS10]. However, none of these works provide guarantees like confidentiality and availability and none considers a cloud-of-clouds.

Some works on cloud storage deal with the high-availability of stored data through replication of this data on multiple clouds, and thus are closely related with the work done in TClouds. The HAIL (High-Availability Integrity Layer) protocol set [BJO09] aggregates 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: it only deals with static data (it is not possible to manage multiple versions of data), it requires that the servers run some code, and does not provide guarantee of confidentiality of the stored data. The RACS system (Redundant Array of Cloud Storage)[ALPW10] employs RAID5-like techniques (mainly erasure codes) to implement high-available and storage-efficient data replication on diverse clouds. It 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.

There are several works about obtaining trustworthiness from untrusted clouds. Depot improves the resilience of cloud storage making assumptions that storage clouds are fault-prone black boxes [MSL+10a]. 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 [FZFF10a] and Venus [SCC+10a] make similar assumptions to implement services on top of untrusted clouds. Wang et al. present a scheme to improve the integrity guarantees of data stored in a cloud, by allowing a third party auditor to verify this integrity on behalf of the data owner [WWL+09]. Santos et al. and Cabuk et al. use trusted computing to ensure the integrity of the infrastructure with the objective of guaranteeing the integrity and confidentiality of data in a cloud [SGR09, CDE+10]. Brugher presents solutions to protect storage clouds from web attacks like SQL injection and cross site scripting by using techniques such as input validation [Bru10]. 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 prevent service outages.

3.3 Computing in Clouds

Besides storing huge amounts of data in clouds, the outsourcing of computing tasks is another fundamental pillar of cloud computing. Being cost efficient especially if the demand for computing power is very unsteady, this approach involves a high risk regarding security of operational data and integrity of the services executed in clouds [csa09].

Whereas data can be stored confidentially by encrypting it, ensuring the integrity of operational data and outsourced computations is a much harder problem. A subtle change in the remote computation, whether caused inadvertently by a bug or deliberately by a malicious adversary, may result in wrong responses to the clients. Such deviations from a correct specification can be very difficult to spot manually.

Suppose a group of clients, whose members trust each other, relies on an untrusted remote server for a collaboration task. For instance, the group stores its project data on a cloud service and accesses it for coordination and document exchange. Although the server is usually correct and responds properly, it might become corrupted some day and respond wrongly. One objective of TClouds is to discover such misbehavior and take compensation measures by leveraging the cloud-of-clouds concept.

A common approach for tolerating faults, including adversarial actions by malicious, so-called Byzantine servers, relies on replication [CBPS10]. However, as long as replicated services are only exported to a single cloud, their reliability and availability depends solely on the reliability and availability of the chosen cloud provider. Distributing replicated services over multiple cloud providers will allow TClouds to offer highly available execution of arbitrary services.

Moreover, TClouds will implement different techniques which can be used to check the consistency of the clouds combined in a cloud of clouds. This can be achieved, for instance, on the basis of authenticated data structures [NN00, MND$^+$04, TT05], a variant of Merkle hash trees for memory checking [BEG$^+$94] generalized to arbitrary search structures on general data sets. Authenticated data structures consist of communication-efficient methods for authenticating database queries answered by an untrusted provider. The two- and three-party models of authenticated data structures allow only one client as a writer to modify the content. In contrast, components of TClouds will allow any client to issue arbitrary operations, including updates.

This entails a multi-writer model as it is addressed, for example, by Mazières and Shasha [MS02]. They introduce untrusted storage protocols and the notion of fork-linearizability (under the name of fork consistency), and demonstrate them with the SUNDR storage system [LKMS04]. Subsequent work of Cachin et al. [CSS07] improves the efficiency of untrusted storage protocols. A related work demonstrates how the operations of a revision control system can be mapped to an untrusted storage primitive, such that the resulting system protects integrity and consistency for revision control [CG09].

FAUST [CKS09] and Venus [SCC$^+$10b] extend the model beyond the one considered here and let the clients occasionally exchange messages among themselves. This allows FAUST and Venus to obtain stronger semantics, in the sense that they eventually reach consistency (in the sense of linearizability) or detect server misbehavior. TClouds proposes a service-integrity verification component that follows a model without client-to-client communication, which makes fork-linearizability, or one of the related "forking" consistency notions [CKS09], the best that can be achieved [MS02].

Several recent cloud-security mechanisms aim at a similar level of service consistency as guaranteed by TClouds. They include the Blind Stone Tablet [WSS09] for consistent and private database execution using untrusted servers, the SPORC framework [FZFF10b] for securing group collaboration tasks executed by untrusted servers, and the Depot [MSL$^+$10b] storage system.

Chapter 4

Initial Architecture Specification

Chapter Author:
Paulo Veríssimo (FFCUL)

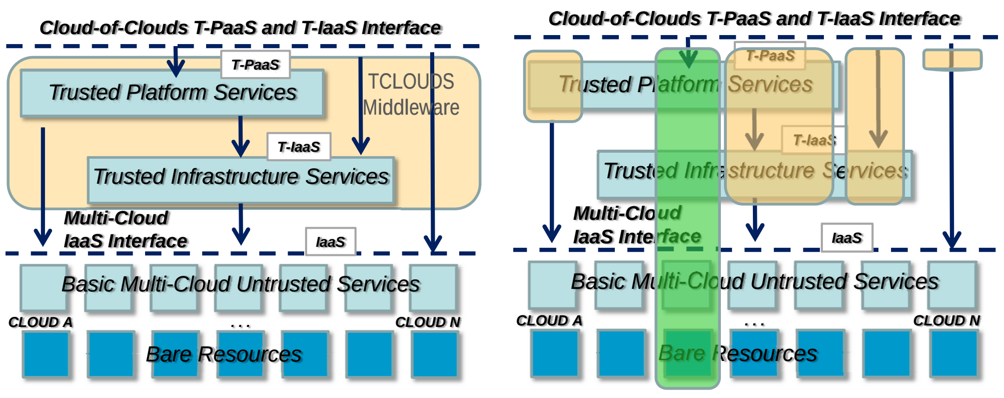
4.1 Initial Architecture Specification

In this section, we present the architecture. We begin by introducing the key aspects of the architecture, contrasting them, when appropriate, with the propositions enumerated earlier. Then, we discuss the main building blocks and enabling components: diverse baseline multi-cloud services; trusted subsystems and mechanisms which induce baseline fault and intrusion prevention; middleware devices that achieve runtime automatic tolerance and protection; middleware protocols promoting intra- and inter-cloud fault and intrusion tolerance. We briefly discuss the placement of the components for adaptive resilience to be presented later in the report. Finally, we present several possible instantiations of the architecture addressing the security and dependability problems of as many realistic cloud-related scenarios.
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, and/or managing a set of replicas, without bothering about network or host intrusion threats.

4.1.1 Overview of the key architecture aspects

The key aspects of the architecture are presented and put in perspective, whenever appropriate, with the propositions enumerated earlier. The TClouds architecture, despite being inspired by previous secure and dependable architectures, such as MAFTIA [VNC+06], for the lower level mechanisms, extends them significantly to attend to the specific challenges of cloud computing infrastructures, for example by identifying such important middleware components as Trusted Infrastructure as a Service (T-IaaS) or Trusted Platform as a Service (T-PaaS), as well as the requirements to implement them. Likewise, the architecture is conceived so as to allow the coexistence of legacy cloud systems with a native TClouds one, or the recursive construction of a TClouds-enabled system by superimposing a resilience layer on top of a legacy cloud system.

As mentioned before, TClouds could be described, in short, as a *resilient cloud-of-clouds infrastructure* providing automated computing resilience against attacks and accidents, in complement or in addition to commodity clouds. This enhanced functionality will be achieved through specialised middleware standing between low-level, basic multi-cloud untrusted services, and the applications requiring security and dependability, as depicted in Figure 4.1.


Cloud-of-Clouds T-PaaS and T-IaaS Interface
T-PaaS
Trusted Platform Services
TCLOUDS Middleware
T-IaaS
Trusted Infrastructure Services
Multi-Cloud IaaS Interface
IaaS
Basic Multi-Cloud Untrusted Services
CLOUD A
...
CLOUD N
Bare Resources
Cloud-of-Clouds T-PaaS and T-IaaS Interface
T-PaaS
Trusted Platform Services
T-IaaS
Trusted Infrastructure Services
Multi-Cloud IaaS Interface
IaaS
Basic Multi-Cloud Untrusted Services
CLOUD A
...
CLOUD N
Bare Resources

(a) (b)

Figure 4.1: TClouds middleware: (a) Services; (b) Use of services

The main characteristics of the architecture should lead to the fulfillment of the set of requirements originating from the propositions made earlier. The TClouds Middleware will be an enabler of these objectives, by providing:

- Support for heterogeneity and openess of individual resilient clouds: (a) promoting economically inclusive clouds ecosystems; (b) improving resilience by enabling diverse fault tolerance schemes. Figures 4.1a and 4.1b convey that message, by illustrating that solutions can be defined by using heterogeneous commodity clouds (where a single-cloud implementation is possible, as a particular case), and combining heterogenous degrees of resilience. Inserting the cloud federation aspect right at the commodity cloud level allows cloud-of-clouds protocols to be built at all layers of the middleware.

- Transparent resilience, with regard to failures of individual clouds, extended seamlessly to cloud federations, as depicted in Figure 4.1a. There, we can see that the TClouds apparatus is foreseen to run over basic (untrusted) cloud services potentially bought from more than one provider, masking individual cloud failures. As explained in Figure 4.1a, 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 be implemented as well by using Multi-Cloud IaaS Interface basic cloud services.

- Incremental resilience, which can be built by the applications according to their needs, by selectively using the services provided at the Cloud-of-Clouds Trusted Interface, with different degrees of resilience, as exemplified in Figure 4.1b: untrusted commodity cloud IaaS services; middleware T-PaaS, Trusted Platform Services; middleware T-IaaS, Trusted Infrastructure Services; advanced native (resilient) TClouds implementations (in green in the Figure), built from scratch by cloud providers (to sell resilient public clouds), or by other companies (to "cloudetise" their IT into private clouds), over their bare resources.

- Modular functions and protocols, to be re-used at the different instantiations of the architecture, namely for the construction of resilient cloud middleware at different levels, satisfying different deployment modes, from software appliances near end users, through middleware based mediators achieving resilience over remotely accessed commodity clouds-of-clouds, to encapsulating local resilience layers over commodity clouds, or native resilient clouds.
- Seamless deployment of the necessary TClouds functions and protocols: between software and hardware based implementations; in single- and multiple-cloud environments; in server-side and client-side implementations.

The last two bullets will depend on implementation strategies for resilience, to be discussed ahead.

4.1.2 Main Building Blocks

The TClouds architecture thus provides applications with a wealth of interfaces to produce incremental resilience solutions with single or multiple clouds: TClouds Trusted Platform services (T-PaaS) on top of the middleware layer; TClouds Trusted Infrastructure services (T-IaaS) from within the middleware layer; Infrastructure services (IaaS) from available commodity untrusted clouds.

The main building blocks of the architecture that implement this functionality, illustrated by Figure 4.2a, are introduced and explained in this section. We discuss the distributed services contributing to the TClouds middleware, namely the placement of the components and services for adaptive resilience to be presented later in the report, alternatively at infrastructure-as-service (IaaS) or platform-as-a-service (PaaS) level.

Basic multi-cloud untrusted services

This block represents the available 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 Services

This building block represents trusted-trustworthy versions of IaaS services, namely storage and processing power. The idea is to offer file 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.

Trusted Platform Services

This building block represents trusted-trustworthy services at a higher level of abstraction, provided through extensions of the resilience layer implemented by the TClouds middleware, built on top of either or both the IaaS and the T-IaaS. These services normally deploy a semantics useful to build complex reliable and distributed applications. Examples are: state machine replication, consistent service execution, etc. Once more, these services are implemented by combinations of fault/intrusion prevention and tolerance mechanisms and protocols, for example, Byzantine fault-tolerant (BFT) protocols.

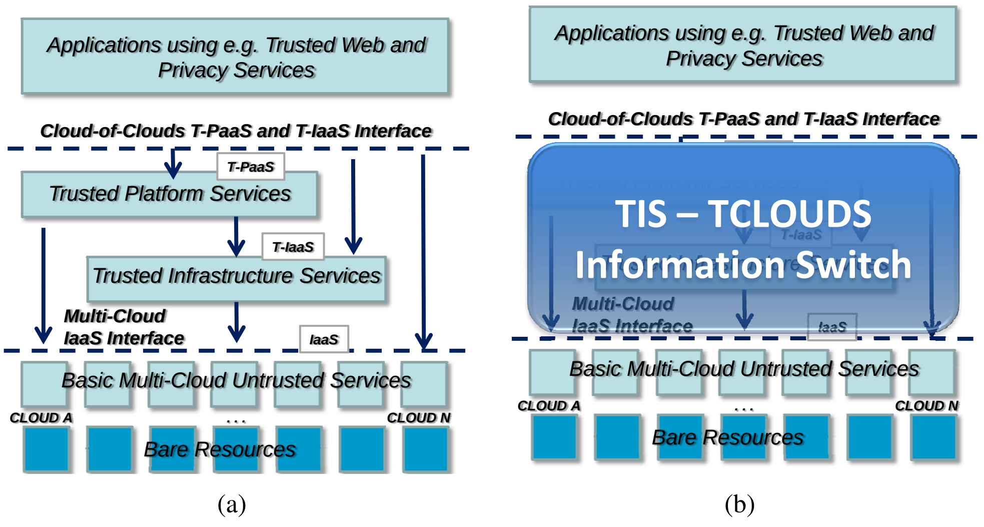

Applications using e.g. Trusted Web and Privacy Services
Cloud-of-Clouds T-PaaS and T-IaaS Interface
T-PaaS
Trusted Platform Services
T-IaaS
Trusted Infrastructure Services
Multi-Cloud IaaS Interface
IaaS
Basic Multi-Cloud Untrusted Services
CLOUD A
...
CLOUD N
Bare Resources
(a)
Applications using e.g. Trusted Web and Privacy Services
Cloud-of-Clouds T-PaaS and T-IaaS Interface
TIS – TCLOUDS Information Switch
Multi-Cloud IaaS Interface
IaaS
Basic Multi-Cloud Untrusted Services
CLOUD A
...
CLOUD N
Bare Resources
(b)

Figure 4.2: TClouds architecture: (a) Block diagram; (b) TIS - TClouds Information Switch

TClouds Information Switches (TIS)

TClouds can either be deployed by final users, third-party added-value providers, or commodity providers wishing to directly offer some form of cloud resilience. To allow seamless deployment of these alternatives, we need to materialize the middleware in a modular way that can morph to each particular configuration of TClouds.

The resilience mechanisms implemented in a distributed way (inter-cloud or, more appropriately, above the commodity or native cloud interface) are essentially based on a conceptual “box”, the *TClouds Information Switch* or TIS, which runs the middleware protocols and mechanisms implementing the resilience components already mentioned, as shown in Figure 4.2b. Each TIS instantiation encapsulates the services in use by that configuration, which are all or part of the services defined in the TClouds architecture. These units, with the adequate configuration and placement, materialize the several TClouds incarnations in a seamless and modular way, as required by the architecture analysis previously made.

TIS implementation depends on the particular incarnation: dedicated machine; fault and intrusion tolerant (FIT) appliance box containing several TIS replicas implemented as virtual machines; may run different sets of functions, depending on location. The TIS can be built with incremental levels of resilience, depending on its criticality.

Trustworthy TIS-TIS interconnection through TClouds communication services secures information flows in the architecture.

TClouds Information Agents (TIA)

The TClouds Information Agent (TIA) can be seen as a particular implementation of a TIS, as a software appliance residing with end clients. Like the TIS, it runs different sets of functions, depending on specific protocols being used on the client side.

TIAs require no additional hardware as a general rule. However, running in the client space, they are subject to a great level of threat. This can be mitigated by configurations where the TIA logic is aware of the existence of minimal additional hardware (e.g., trusted components) to improve its resilience. On the other hand, the TIA option requires client modifications to achieve the desired TClouds functionality.

Whenever needed, trustworthy TIA-TIS interconnection through TClouds communication services secures information flows in the architecture.

4.2 TClouds deployment alternatives

In this section, we discuss how TClouds addresses several realistic resilient cloud computing (CC) scenarios, by presenting as many deployment alternatives. This objective will be attained by offering the designer different instantiations of the architecture, addressing the security and dependability problems put by each scenario. To keep complexity to a manageable level, the TClouds architecture should serve these objectives essentially by re-using and reconfiguring the same basic components providing trusted IaaS and PaaS services.

In particular, we discuss implementations of TClouds functionality preserving the use of legacy commodity clouds IaaS, either by resorting to client-side software, or to server-side software. The latter can alternatively be implemented by sub-systems located at an added-value trusted-clouds provider offering managed resilient cloud services, or in-house, as a trusted gateway to commodity cloud services, providing cheap solutions for achieving trusted cloud services. An interesting step of the in-house solution is studied, in the form of a private TCLOUD encapsulating the organisation IT. Finally the architecture 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 to TClouds: TClouds-enabled Cloud preserving data center machinery and software, and adding a resilience layer on top; TClouds native Data Center and Cloud, built from scratch with TClouds mechanisms.

TClouds-enabled client-resident software

Depicted in Figure 4.3, client-resident software is composed of add-on modules allowing direct implementation of some secure services over commodity clouds, by organisations' clients.

It is the simplest TClouds implementation, not requiring additional machinery, but implying modifications in all client machines wanting to access resilient cloud services.

TClouds-enabled cloud services mediator (client-side)

Middleware layer local to an organisation is shown in Figure 4.4, using TClouds protocols and mechanisms as front-end of one or more commodity clouds infrastructures, centralising the provision of resilient services to the whole organisation.

Technically, this can be seen as a generalisation of the client-resident software alternative, with dedicated machines which are "clients" to the cloud providers, and simultaneously "servers" to the organisation's end-clients.

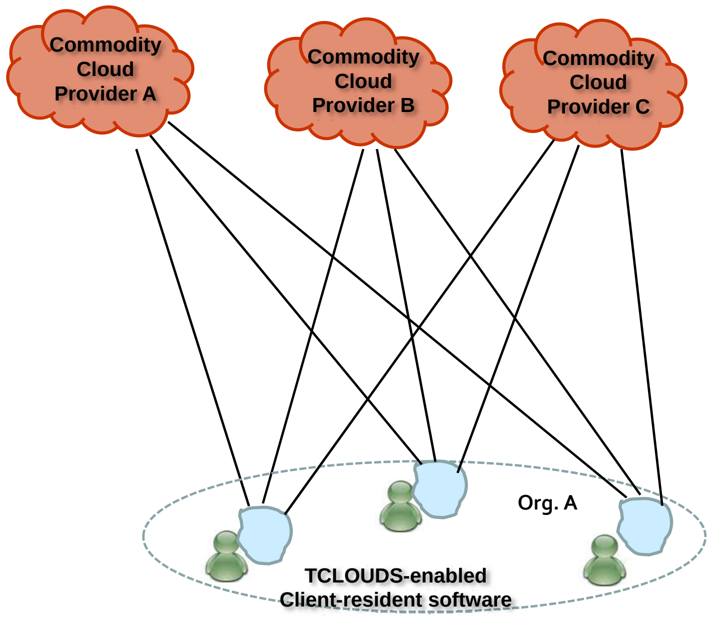

Commodity Cloud Provider A
Commodity Cloud Provider B
Commodity Cloud Provider C
Org. A
TCLOUDS-enabled Client-resident software

Figure 4.3: TClouds-enabled client-resident software

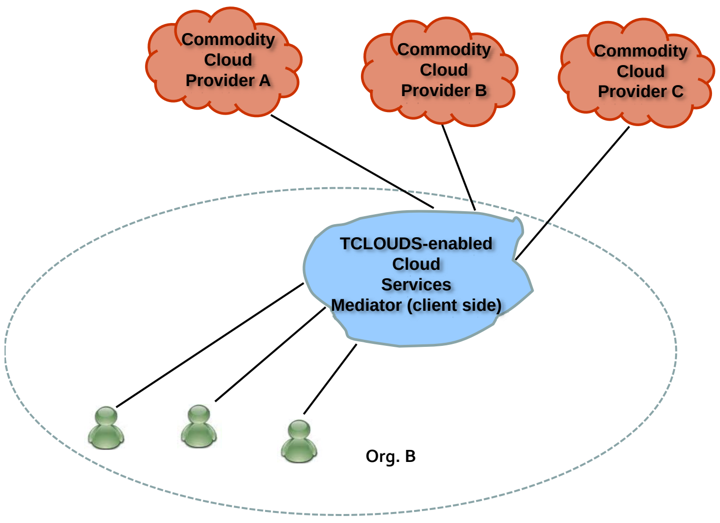

Commodity Cloud Provider A
Commodity Cloud Provider B
Commodity Cloud Provider C
TCLOUDS-enabled Cloud Services Mediator (client side)
Org. B

Figure 4.4: TClouds-enabled cloud services mediator: client-side

TClouds-enabled cloud services mediator (server-side)

As an alternative, Figure 4.5 illustrates a middleware layer in a stand-alone organisation, using TClouds protocols and mechanisms as front-end of one or more commodity clouds infrastructures, posing itself as an added-value clouds provider supplying resilient services.

Technically, it can be seen as a generalisation of the client-side mediator alternative, where the relevant machines provide services to several end-client organisations.

TClouds-enabled cloud provider

Infrastructure local to a cloud provider, using TClouds protocols and mechanisms encapsulating a legacy commodity clouds infrastructure, is shown in Figure 4.6a, as an easy path to in-house transformation of untrusted clouds into resilient ones, by the cloud provider industry.

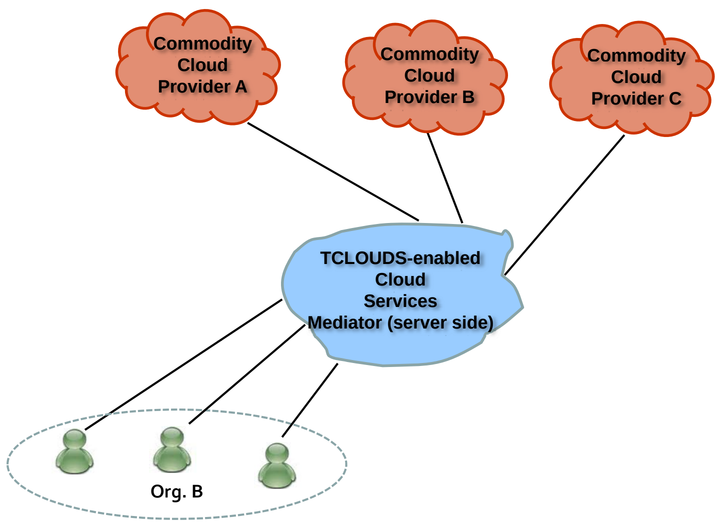

Commodity Cloud Provider A
Commodity Cloud Provider B
Commodity Cloud Provider C
TCLOUDS-enabled Cloud Services Mediator (server side)
Org. B

Figure 4.5: TClouds-enabled cloud services mediator: server-side

Technically, it can be seen as a specialisation of the server-side mediator, where the relevant machines are now closely coupled to the legacy cloud data center machinery of a single cloud provider.

The trustworthiness/performance product may increase, at the loss of diversity.

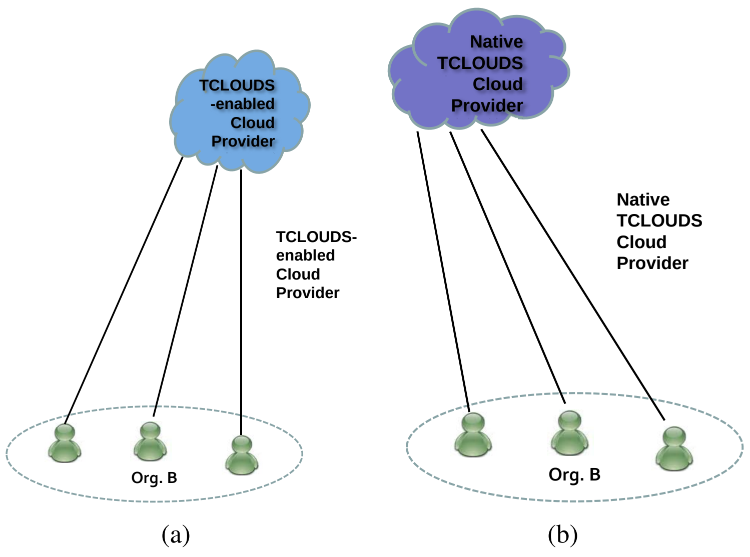

TCLOUDS -enabled Cloud Provider
TCLOUDS- enabled Cloud Provider
Org. B
(a)
Native TCLOUDS Cloud Provider
Native TCLOUDS Cloud Provider
Org. B
(b)

Figure 4.6: Cloud provider solutions: (a) TClouds-enabled cloud provider; (b) Native TClouds cloud services provider

Native TClouds cloud services provider

The utimate step is shown in Figure 4.6b, where we can see an infrastructure local to a cloud provider, using native TClouds protocols and mechanisms in the design of the data centers from scratch.

This alternative is bound to achieve the best trustworthiness/performance product, that is, ultimately trustworthy T-IaaS and T-PaaS for a single cloud provider, but at the cost of losing diversity.

Diverse TClouds ecosystem

The final and overall goal is to be able to supply a cloud system architect with all the alternatives above.

This rich infrastructure, depicted in Figure 4.7, will be achieved by the eventual availability of the several alternatives discussed. It prefigures a true ecosystem capable of offering the best possible tradeoffs to clients and providers of resilient cloud services, either end-clients or mediators.

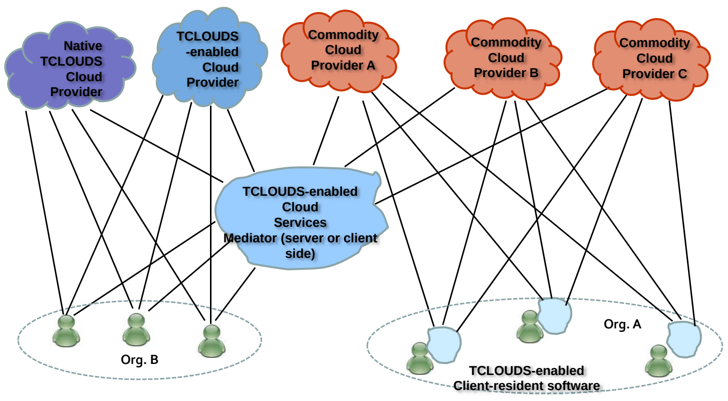

Native TCLOUDS Cloud Provider
TCLOUDS -enabled Cloud Provider
Commodity Cloud Provider A
Commodity Cloud Provider B
Commodity Cloud Provider C
TCLOUDS-enabled Cloud Services Mediator (server or client side)
Org. B
Org. A
TCLOUDS-enabled Client-resident software

Figure 4.7: Diverse TClouds ecosystem

4.3 Adaptive Resilience

Incremental levels of resilience may be obtained both at micro (local node and intra-cloud) and macroscopic level (inter-cloud), by the definition of tradeoffs between resilience and cost or complexity of solutions.

4.3.1 TClouds local node architecture

Trusted-trustworthy operation [VNC+06] is an architectural paradigm whereby components prevent the occurrence of some failure modes *by construction*, so that their resistance to faults and hackers can justifiably be trusted. Given the severity of threats expected, some key components are built using architectural hybridization methods in order to achieve extremely high robustness. In other words, some special-purpose components are constructed in such a way that we can argue that they are always secure, so that they can provide a small set of services useful to support intrusion tolerance in the rest of the system. This concept is in line with, but richer than, technological concepts like trusted computing or trusted platform modules.

Transparent fault and intrusion tolerance aims at preserving the illusion of a standard cloud system to applications, hiding the complexity of building resilience from the latter. It is achieved by specific replica control and communication algorithms to be developed.

Some resilience mechanisms will be achieved at the node level, through the use of diverse security and dependability mechanisms, such as: trusted-trustworthy components like TPMs or wormwholes; VM replication and recursivity; self-healing, etc. A local node architecture is

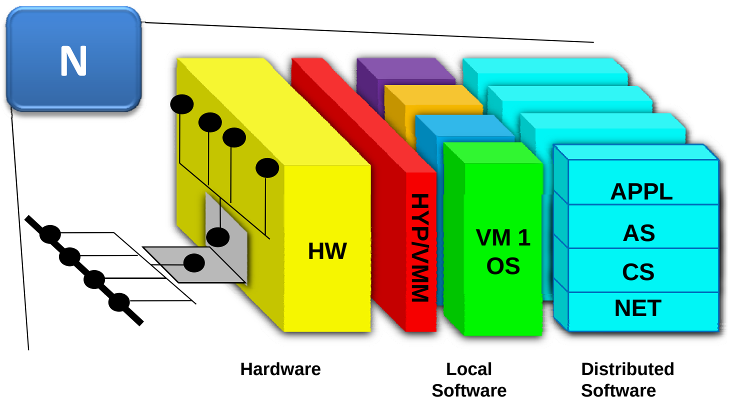


Figure 4.8: Local architecture of a TClouds node

proposed that allows implementing these solutions in an effective and legacy-preserving way. A snapshot of the TClouds architecture detailing a node and its interconnection methods is depicted in three dimensions in Figure 4.8, where we can perceive the following node structuring notions, which closely follow the node structuring principles for intrusion-tolerant systems explained in [VNC+06]:

- The notion of *trusted hardware* – as opposed to an untrusted one. For example, most of the hardware of a TIS is considered to be untrusted, with small parts of it being considered trusted-trustworthy.

- The notion of *trusted support software*, trusted to execute a few critical functions correctly, the rest being subjected to malicious faults.

- The notion of *run-time environment*, offering trusted and untrusted software and operating system services in a homogeneous way.

- The notion of *trusted distributed components*, for example software functions implemented by collections of interacting TIS middleware.

In the context of this chapter, we will focus on the TClouds Information Switch (TIS) nodes. However, other specific nodes, for example, native TClouds data center nodes needing to meet high trustworthiness standards, may be also built to a similar structure.

Firstly, there is the *hardware* dimension, which includes the node and networking devices that make up the physical distributed system. As said before, we assume that most of a node's operations run on untrusted hardware, e.g., the usual machinery of a computer, connected through the normal networking infrastructure. However, some nodes– TIS, for example– may have pieces of hardware that are trusted, i.e., where by construction intruders do not have direct access to the inside of those components. The types of trusted hardware featured in TClouds may include standard TPMs, or dedicated *appliance boards with processor*, plugged into the node's main hardware.

Secondly, services based on the trusted hardware are accessed through the *local support* services. The rationale behind our trusted components is the following: whilst we let a local node be compromised, we make sure that the trusted component operation is not undermined (crash failure assumption).

Thirdly, there is the *distributed software* provided by TClouds: middleware layers on top of which distributed applications run, even in the presence of malicious faults (far right in Figure 4.8).

4.3.2 Incremental node resilience strategies

Given that different application and systems will require different levels of trust, the architecture must allow for an incremental range of resilience solutions, in the interest of the best tradeoff with performance, cost, or complexity.

A key issue is the resilience of the TClouds nodes (TIS) against direct attacks. We give a few examples illustrating the possible TIS construction methods, to achieve the desired incremental range of resilience:

Ruggedised simplex - single ruggedised machine;

Loosely coupled duplex or N-plex - replicated loosely in the network;

Closely coupled N-plex - replicated with a private broadcast network (TMR-like);

Tightly coupled N-plex - replicated and diverse VMs in a same box;

Twin quad - 2 replicas of VM quads to guarantee BFT + availability.

We recall that these construction alternatives for resilience can be used either in TIS or in machines included in native TClouds instantiations.

4.3.3 Making TIS resilient

Let us briefly discuss how TIS are made trusted-trustworthy components. TIS are built with a combination of untrusted and trusted hardware of varying degrees, depending on the needs and criticality of the services they support. TIS individual resilience can be enhanced by proactive resilience mechanisms providing self-healing, using a construct called Proactive Resilience Wormhole [SNV05], aiming at providing for perpetual execution of a given set of TIS, despite continued intrusion and/or failure of an assumed simultaneous maximum number of TIS at an assumed maximum rate.

These notions can be recursively used to construct a logical TIS which is in fact a set of replicated physical TIS units, running some internal intrusion-tolerant protocols so that the whole appears to the protocol users as a single logical entity, but is in fact resilient to attacks on the TIS themselves. This is a powerful combination since the resilience of protocols running on such intrusion-tolerant TIS components is commensurate to arbitrary-failure counterparts.

This implementation can be simplified, by having for example the TIS be made of a single hardware box, containing several TIS replicas implemented as virtual machines.

4.3.4 TClouds distributed middleware devices

We start by reviewing the TClouds architecture deployment alternatives:

- TClouds-enabled client-resident software
- TClouds-enabled cloud services mediator (client-side)

- TClouds-enabled cloud services mediator (server-side)
- TClouds-enabled Cloud Provider
- Native TClouds cloud services provider
- Diverse TClouds-enabled ecosystem

Satisfying these alternatives requires a few classes of TClouds distributed middleware devices, which we preliminarily discuss.

Byzantine-resilient protocols

The workhorse of the solutions for achieving resilient cloud services will be the so-called Byzantine fault-tolerant protocols, or BFT protocols.

The basic running environment for these modular protocols will be in the form of software modules located with end-clients, e.g., to address the TClouds-enabled client-resident software alternative.

As shown in Figure 4.9, these protocols may involve communication between modules. However, the dotted lines suggest possible alternatives where clients do not communicate with one another, but directly with the several commodity clouds used.

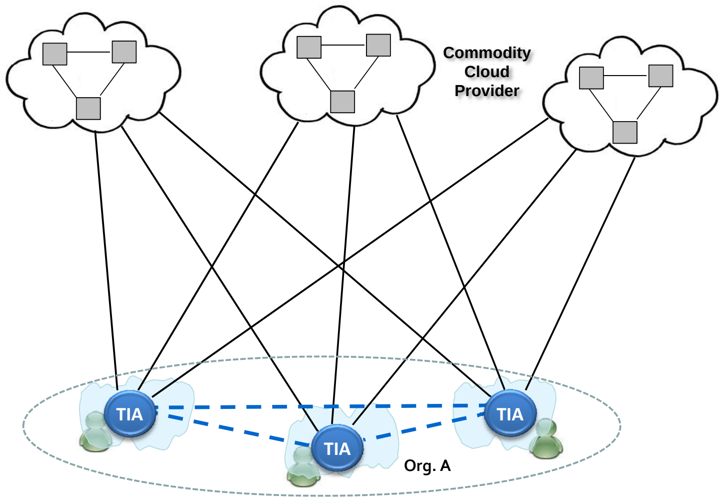


Figure 4.9: Byzantine-resilient protocols

Appliance-box implementation

One simple and compact way of getting TClouds technology running is through a configuration based on a virtualised node running several TIS instantiations, amongst different functional units and replicas. This appliance box can for example, be used to enable resilient cloud services in an SME which has simple IT resources. As Figure 4.10 suggests, the appliance box runs the TClouds middleware, interfaces the basic multi-cloud ecosystem and transparently supplies the organisation clients a resilient clouds interface.

Note that the protocols discussed in the previous section can modularly be re-used within the appliance-box's TIS.

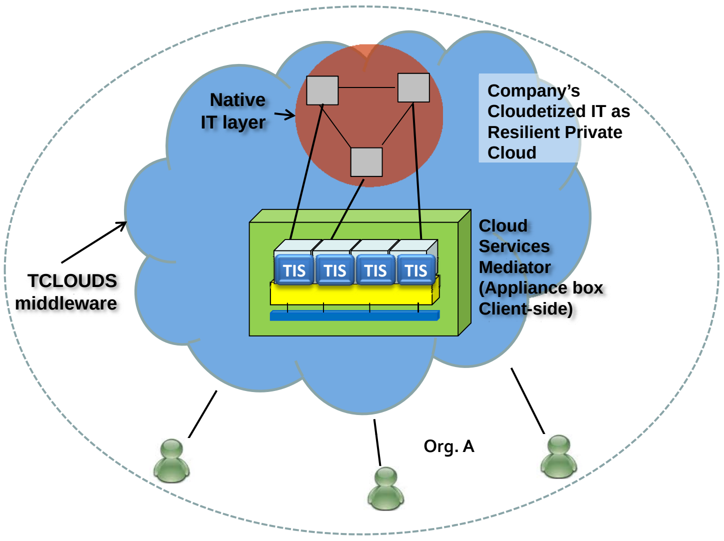


Figure 4.10: TClouds Appliance box implementation

Figure 4.11 suggests how to implement a private cloud with a TClouds appliance box. The bare IT resources of the organisation are cloudetised, wrapped by the TClouds middleware, resident in the appliance box, and offered as resilient cloud services to the organisation users.

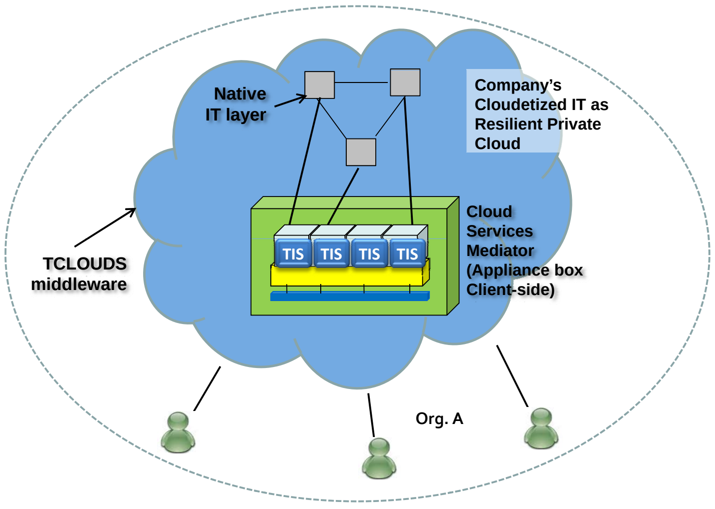


Figure 4.11: Implementing a private cloud with a TClouds appliance box

Server-set implementation

A powerful alternative is given by considering the implementation of potentially the same protocols run in the appliance box, by an actual set of distributed servers. The advantage is increased processing power and greater failure independence. With this configuration, it is tecnically feasible for a mediator, an added-value cloud 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 4.12 illustrates.

The downside may be that this configuration requires trust in a single cloud provider, by the end users. This aspect can be mitigated by two arguments: (a) given the resilience measures we foresee to implement TISs, the greater failure risk lies with the raw IaaS from the commodity providers, mitigated by the diversity of the untrusted multi-cloud ecosystem; (b) the added-value provider may be able to provide objective evidence of its measures to reduce the risks of single-point-of-failure, or vendor lock-in. At least in the early phases of the creation of a trusted clouds ecosystem, we believe this to be an interesting technological alternative, for example for SMEs.

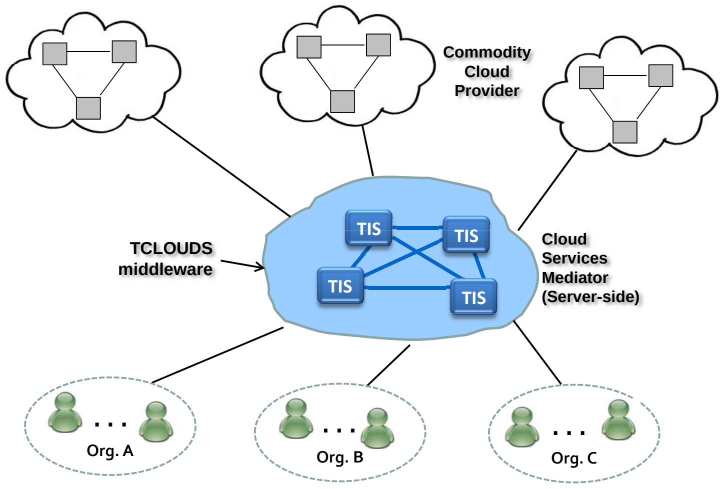


Figure 4.12: Server-set implementation

TIS-implemented 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 by added-value providers will create a push for open solutions, of the king offered by TClouds, which we discuss in this section.

The first and easiest way for a commodity provider to offer resilient services is to preserve its 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 cloud's data centers, as shown in Figure 4.13a. Note that the TIS are shown as simplex, for simplicity, but they may be constructed to be as resilient as desired, according to the strategies discussed in Section 4.3.2.

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, as discussed in Section 4.3.1.

As the Figure 4.13b suggests, a TClouds native cloud is achieved by re-implementing the data centers to be TClouds compliant (DC_T). This means at least two things: (a) the use of local fault/intrusion prevention and tolerance mechanisms to enhance the basic machines and resources; (b) the re-design of the BFT protocols used to implement the TClouds middleware, to run embedded in the bare resources, making them 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.

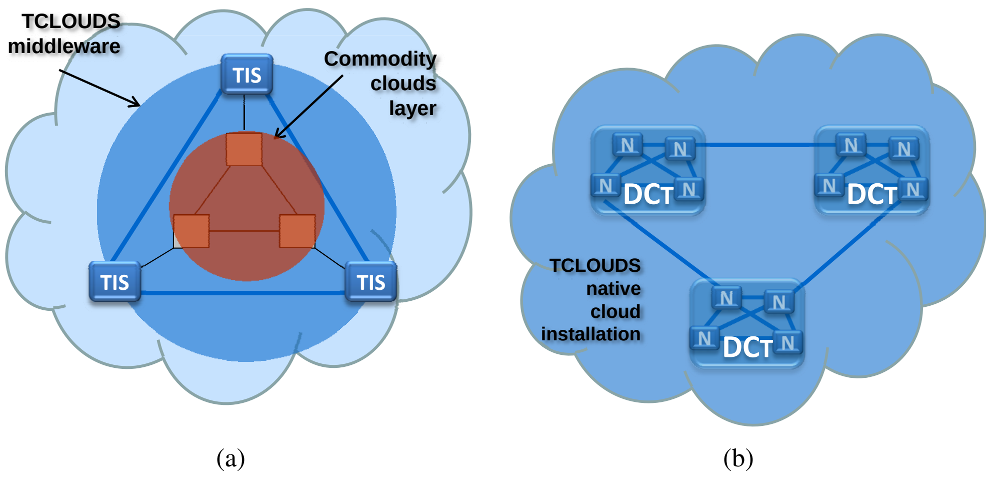


Figure 4.13: Commodity provider evolution: (a) TClouds-enabled cloud; (b) TClouds native Data Center and Cloud

Chapter 5

Components for Adaptive Resilience

Chapter Authors:
Alysson Bessani (FFCUL), Marcelo Pasin (FFCUL), Christian Cachin (IBM), Johannes Behl (FAU), Klaus Stengel (FAU) and Davide Vernizzi (POL)

In this chapter we briefly present the TClouds trusted services presently being considered, which will be detailed later in the report (a chapter each).

5.1 Object Storage

Many current cloud providers offer object storage services similar to Amazon's Simple Storage Service (S3). These services are used to store large data blobs, identified with a unique name. IBM and FFCUL cooperate on providing subsystems for reliable and secure blob storage through a federation of object storage services from multiple providers.

Software is offered under the form of a library, linked to each client's code, and executed before it accesses cloud storage. Management and setup are performed the same way as for accessing one single storage provider, and the library does not require client-to-client communication.

Multiple clients may concurrently access the same remote storage provider and operate on the same objects. It is possible thanks to an interface that contains the basic and most common operations of object cloud storage (read, write, remove). As all cloud storage providers offer similar interfaces, the one proposed by IBM and FFCUL uses their common denominator.

The storage system provides confidentiality through encryption, integrity through cryptographic data authentication, and reliability through data replication and erasure coding. Key management for encryption and authentication keys is integrated.

Cloud-of-clouds object storage is very simple architecturaly. It relies on practicaly no other building block besides the currently existing vanilla cloud storage services. Object storage can be pictured in TClouds architecture as in Figure 5.1. A TIA represents the library (or a proxy) in the client-side that uses services from multiple cloud providers give the client the illusion of using a single service. This subsystem is detailed in chapter 6.

5.2 Consistency for untrusted service execution

Several tools exist today which allow a single user to verify the integrity and availability of his own data stored in the cloud. But when multiple users access the same data, they cannot guarantee integrity between a writer and multiple readers. And when the cloud should deliver a more complex service than storage and retrieval, such methods must be much more elaborate.

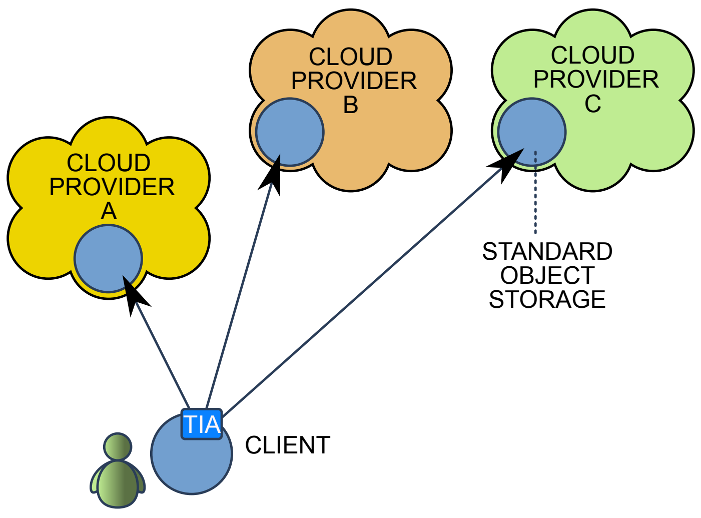


Figure 5.1: TCLODS object storage mapped onto its architecture.

Digital signatures may be used by a client to verify integrity of data created by others. Using this method, each client needs to sign all his data, as well as to store an authenticated public key of the others or the root certificate of a public-key infrastructure in trusted memory. This method, however, does not rule out all attacks by a faulty or malicious service. Even if all data is signed during write operations, the server might omit the latest update when responding to a reader, and even worse, it might "split its brain," hiding updates of different clients from each other.

Some solutions are to use trusted components in the system, which allow clients to audit the server, guaranteeing atomicity even if the server is faulty. But without additional trust assumptions, the atomicity of all operations in the sense of linearizability cannot be guaranteed. Though a user may become suspicious when he does not see any input from a collaborator, the user can only be certain that the server is not holding back information by communicating with the collaborator directly; such user-to-user communication is indeed employed in some systems for this purpose.

The component described here aims at discovering such misbehavior, in order for the users to take some compensation action against a faulty server. Ensuring the *integrity* of remote data and outsourced computations in this way will benefit the users directly and enhance their trust into the service.

The integrity verification component described in Chapter 7 guarantees atomic operations to all clients when the provider is correct and fork-linearizable semantics when it is faulty; this means that all clients which observe each other's operations are consistent, in the sense that their own operations, plus those operations whose effects they see, have occurred atomically in same sequence. This protocol generalizes previous approaches that provided such guarantees only for outsourced storage services to arbitrary services.

5.3 State Machine Replication

Besides security, other measures are necessary to prevent intrusions. This becomes true mainly because of the complexity of the clouds, where programming errors will potentially exist forever. A secure system can be deceived by exploiting its known defects. Because of that, specific measures that allow for tolerating intrusions must also be addressed when building the trustworthy clouds. State machine replication is a well known technique used to build intrusion-tolerant systems, by considering intrusions as Byzantine faults.

Server and client are the basic structures used to implement distributed systems, and clouds are not different. Servers offer services and clients use such services by invoking them. An invocation is done by the client, sending a request message to the server, which sends a reply message to the client with the corresponding results.

Fault-tolerant distributed systems are implemented by replicating the components prone to failures, so they can fail independently without compromising the service availability. A practical way of dealing with intrusions is to model the system as being fault-tolerant, capable of keep working well even under failures. With Byzantine failures, a component is allowed to fail in arbitrary ways, including the most common stop and crash failures, but also processing requests incorrectly, corrupting their local state, or producing incorrect or inconsistent outputs, typically found in intruded software.

Byzantine fault-tolerant services are implemented using replicated state-machines, that upon receiving a request deterministically change to a new state and send a reply. All state-machine replicas start with the same state and requests are sent to them using reliable, ordered, broadcasts from clients. Majority (voting) is used in the clients to select the correct reply among those from all replicas. A system built using state-machine replication ensures *availability* by exploiting replication and diversity to run the replicas of the service on several clouds, thus allowing access to it as long as a subset of them is reachable. It also ensures *integrity* of the service executed as long as the majority (at least) of the clouds are correct and run the correct service code.

To minimize the probability of correlated failures and improve the fault- and intrusion-tolerance of a replicated set of services, we expect them to be deployed on different operating systems, Java virtual machines and hypervisors. It allows to avoid common software faults and shared vulnerabilities. By hosting replicas in different clouds (or in different zones of the same cloud) we can ensure that local outages and security events do not affect more than one replica.

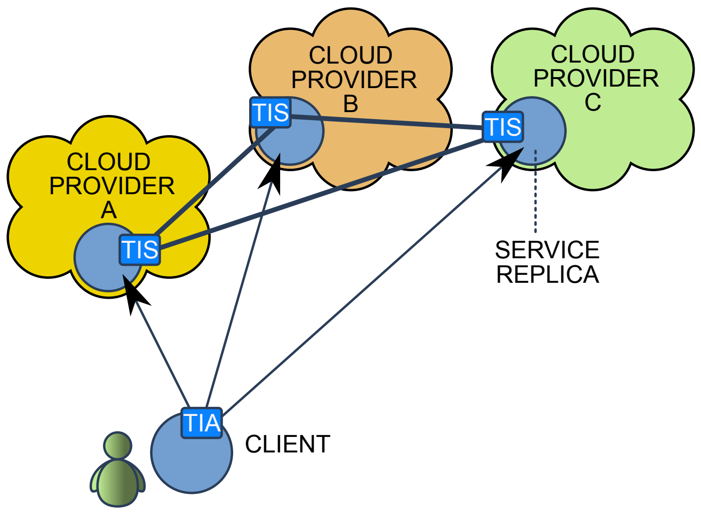


Figure 5.2: State-machine replication mapped onto the TClouds architecture.

A projection of state machine replication to the TClouds architecture is diplayed in Figure 5.2. A TIA represents the library (or a proxy) in the client-side that invokes multiple replicas and give the client the illusion of invoking just one. The TISs represent the libraries attached to the state-machine replicas, that give to the state-machine the illusion of being ivoked alone, whereas multiple replicas are invoked at the same time. A much deeper presentation on state-machines and the implementation being proposed as a subsystem of TClouds is given in Chapter 8.

5.4 Fault-tolerant workflow execution

Following the slogan "Everything as a service", cloud computing leads to a constantly growing number of services, ranging from very basic infrastructure offerings such as computing power or storage space, to more complex platform services like application and other execution environments, to entire software solutions, for example, Web-based office tools. Their growing number comes along with a growing demand for a common way to coordinate and manage interactions of all these services. As the majority of them can be accessed by standard Web-service technology, this demand is basically the demand for orchestrating Web services, a problem for which exactly the field of *business process management* offers solutions. For instance, infrastructures for the *Business Process Execution Language* (BPEL) provide a platform for executing business processes, or more general, workflows based on multiple simpler Web services. Offered as Web services themselves, these complex workflows are also called *composite Web services*.

Most of the generic or domain-tailored solutions for creating, executing and managing composite Web services exhibit sophisticated interfaces, a multitude of connectors to subsystems, and increasing support for non-functional properties such as scalability and security. Nevertheless, fault tolerance has received relatively limited attention so far. Standard *BPEL engines*, responsible for executing composite Web services within BPEL infrastructures, log state changes to persistent storage to enable the recovery of active business processes after a reboot or crash. Besides slowing down the execution speed during normal operation, the reliability of this mechanism depends on the reliability of the storage. In addition, BPEL provides only limited means to handle failures of the Web services the workflows are based on. Making these Web services fault tolerant is not supported at all by standard BPEL infrastructures. Moreover, the recovery mechanism does not provide any opportunity to tolerate arbitrary faults like malicious attacks or hardware errors. However, recent studies on cloud offerings and hardware in general show, that clouds are less reliable than traditional data centers and hardware failures are more common than previously assumed. This basically inhibits outsourcing of critical processes like financial or medical services to the cloud.

Therefore, the subsystem described in Chapter 9 offers highly available, fault-tolerant execution of critical Web-service–based workflows as a platform service within clouds. The presented solution is practice-oriented, since current BPEL infrastructures and workflows can be widely reused, and it is extensively configurable as well as dynamically adaptable through the use of external coordination services provided by today's cloud infrastructures.

In contrast to other works in this field, the architecture of the subsystems provides fault tolerance by means of active replication at both the process level and the service level. Although the current implementation only tolerates crashes, using active replication is the first step towards tolerance of arbitrary faults. In the presented solution, replication is realized transparently to workflows described by means of BPEL. This is achieved by automatically transforming such workflows into a replication-aware version. Thereby, custom proxy components can intercept all Web-service invocations, which enables them to carry out tasks necessary for replication. This approach also permits to reuse standard BPEL engines, simplifying the implementation. To simplify it further, we heavily make use of Apache ZooKeeper, a service to coordinate distributed applications. This allows externalizing coordination tasks and realizing global configuration as well as dynamic adaptation to changing environmental conditions.

5.5 BFT MapReduce

MapReduce is a framework developed by Google for processing large data sets [DG04]. It is composed by a programming model and a runtime system, being used extensively by Google in its datacenters to support core business functions such as index processing for its web search engine. Its programming model is based on map and reduce functions as found in functional programming (with a somewhat modified meaning). It runs using a large number of computers as found in clusters and datacenters.

Programmers specify map and reduce functions: the former is used to process an input file and generate key-value pairs, the latter is used to merge several such pairs (with the same key) into a single key-value pair. The running environment first splits the input file, then feeds several instances of the map function with those splits. The multiple map outputs are then sorted key-wise and splitted again, now fed to multiple reduce functions, in a phase known as shuffle. Multiple reduce outputs are finally concatenated in an output file.

Under the MapReduce model, a job is easily automatically partitioned into tasks by splitting its input data. Having a large number of independent tasks allow for shrinking and stretching the footprint of a job during its execution in a MapReduce environment. A job can use up to any number of resources, it just takes longer to end when using less resources. This characteristics make MapReduce (model and enviroment) a very suitable platform for cloud computing.

Users submit jobs to MapReduce containing map and reduce functions and an input file. The input file is stored in a special file system that breaks it into smaller replicated pieces, called splits, homogeneously stored in the same nodes available for running jobs. Jobs are also broken into pieces, called tasks. Every split is fed to a different map task, preferably executed in the node that contains the split (avoids to transfer the split). When all map tasks are ended, their output maps are sorted and hashed into partitions, one for every reduce task. The partitions are transfered to the corresponding reduce tasks, which reduce data producing outputs that are concatenated into a file.

The solution proposed by FFCUL for intrusion-tolerant MapReduce includes replication of all tasks. Every map is produced multiple times and a vote is done in the reduce tasks, before they use their input. Every reduction is produced multiple times and is stored in multiple output files, the vote being done by the user. Instead of implementing this solution by adding TIS or TIA blocks to an unmodified, non-intrusion-tolerant, implementation, we decided to modify a freely available version of MapReduce by adding to it the necessary features for replication and voting, as described in Chapter 10. The result is a complete middleware set that is intrusion-tolerant and distributed by itself, as a native TClouds implementation of a MapReduce PaaS (platform-as-a-service).

5.6 Logging

In our vision of cloud, the presence of a efficient logging system is necessary. In this context we define the Log Service, a service which has as objective to track and to log events that happen in the cloud at different levels.

The main focus of Log Service is to log and track events originating at the infrastructure layer. This service is mainly based on the scheme for secure logging proposed by Schneier and Kelsey in [SK99]. Logs are composed by log entries which are usually small pieces of data, created at high rate and rarely deleted. For these reasons, the Log Service must be capable of recording many log entries and of providing a view on a subset of the log entries that satisfy a particular query. Moreover, in order to ensure the security of the log entries, the Log Service must be capable to guarantee their integrity and confidentiality. Since the log entries may contain sensitive information about the usage of a certain system, Log Service must be capable to mediate every access in order to prevent leakage of information. Therefore the presence of an access control system is required.

Log Service may be considered as the ensemble of three components. The *Log Core* which is the main component and acts as service controller, the *Log Storage* which manages the storage of the log entries and the *Log Console* which acts as public access interface to the Log Service. We foresee different multi-cloud scenarios that represent possible steps along evolutionary paths from a service entirely confined within a single cloud (described in D2.1.1 and briefly recalled in Section 11.2) – the *TClouds* cloud – to a completely distributed one.

As starting point, the LS can be extended to the Log as a Service by moving the location of the nodes originating the log events from the TClouds cloud to a remote one. A possible example could be a remote *private* cloud that wants to outsource the management and the storage of the log events originated at the infrastructure layer to an external cloud entirely dedicated to this purpose.

Another possibility is to track and log Cloud-of-Cloud events, for instance events generated by BFT protocols. In this scenario we consider a Cloud-of-Cloud which is composed by one trustworthy cloud (TCloud) and less trusted clouds. In this architecture, each event generated by the BFT protocols implies the exchange of a large amount of messages between nodes which may be part of different clouds. These messages may be used as trigger for the identification of a certain event. This way, by defining a proper communication protocol among Cloud-of-Cloud nodes, it is possible to realize an event-driven logging system.

Finally, it is possible to consider advanced scenarios in which the Log Service not only runs on the trustworthy cloud, but it can be also distributed on different clouds. The "distributed" aspect of Log Service means that some components like Log Core (or Log Storage) can be replicated on a different cloud nodes (which may be part of untrusted clouds) in order to build, for instance, a fault tolerance system. In this case, special attention must be paid in order to protect the cryptographic keys used to ensure integrity and confidentiality of log entries, especially for those keys which are used on the nodes of the untrusted cloud.

Chapter 6

Object Storage

Chapter Author:
Alysson Bessani (FFCUL)

6.1 Introduction

In this chapter 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 by invoking operations in several individual clouds. More specifically, DEPSKY addresses four important limitations of cloud computing for data storage in the following way:

Loss of availability: temporary partial unavailability of the Internet is a well-known phenomenon. When data is moved from inside of the company network to an external datacenter, it is inevitable that service unavailability will be experienced. The same problem can be caused 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 data.

Loss of privacy: the cloud provider has access to both the data stored in the cloud and metadata like access patterns. The provider may be trustworthy, but malicious insiders are a wide-spread security problem. 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 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 two times the cost of using a single cloud. This seems to be a reasonable cost, given the benefits.

The key insight here is that the 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 code [CT06, GWGR04, MR98a, MR98b, 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 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 data stored in each cloud.

In summary, the main contribution of this chapter is the description of 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.

6.2 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 [tcl10].

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 storage systems, power systems have historical data databases used to store the data collected from the power system. Of course, in a system like this, 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.

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 6.3) 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.

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. Systems like databases, file systems, objects stores and key-value databases can use the cloud as storage layer as long as caching and weak consistency models are used to avoid paying the price of cloud access on every operation.

6.3 The DepSky System

This section presents the DepSky system. It starts by presenting the system architecture, then defines the data and system models, the two main algorithms (DepSky-A and DepSky-CA), and a set of auxiliary protocols.

6.3.1 DepSky Architecture

Figure 6.1 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).

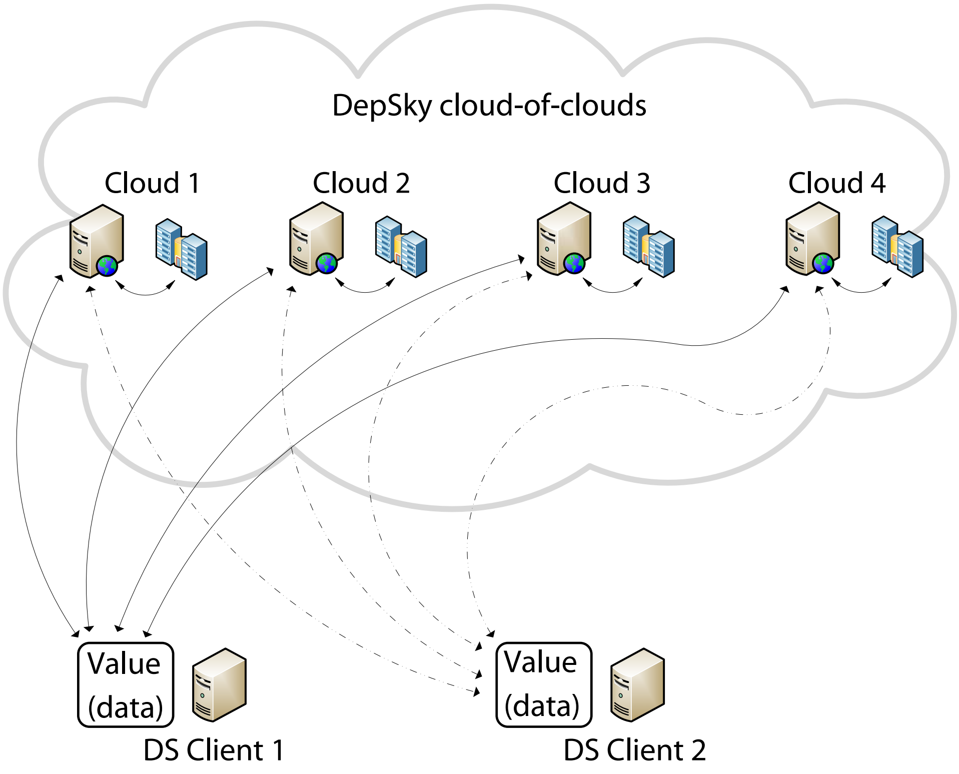


Figure 6.1: Architecture of DEPSKY (w/ 4 clouds, 2 clients).

6.3.2 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 6.3.2 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, MR98a]). 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 informations 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.).

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.

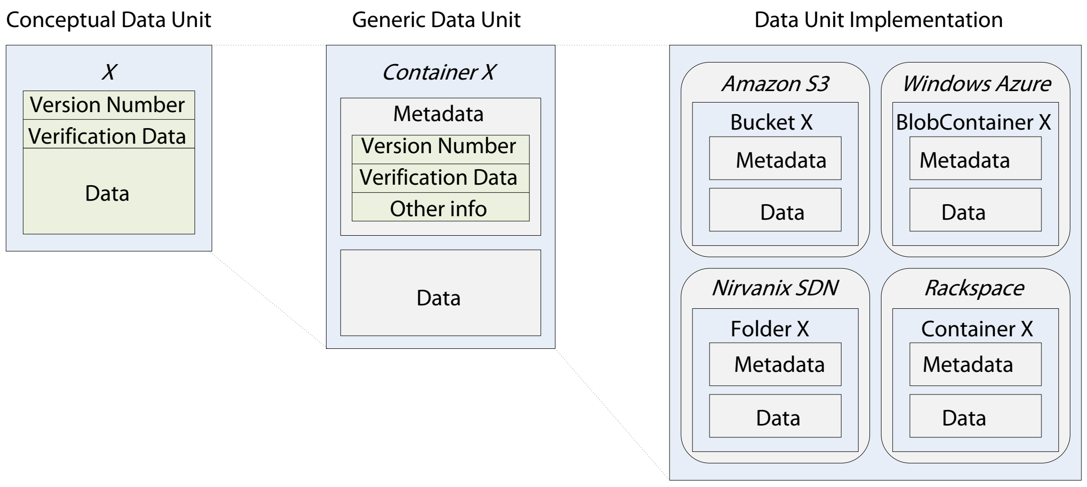


Figure 6.2: DEPSKY data unit and the 3 abstraction levels.

6.3.3 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 6.1, while writers and readers are roles of the clients, not necessarily different processes.

Readers and writers. Readers can fail arbitrarily [LSP82b], i.e., they can crash, fail intermittently and present any behavior. Writers, on the other hand, are only assumed to fail by crashing. We do not consider that writers can fail arbitrarily because, even if the protocol tolerated inconsistent writes in the replicas, faulty writers would still be able to write wrong values in data units, effectively corrupting the state of the application that uses 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.

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 system in order to ensure that readers are only allowed to invoke the list and get operations.

Since we do not trust clouds individually, we assume they can fail in a Byzantine way: the data stored can be deleted, corrupted, created or leaked to unauthorized parties. This is the most general fault model and encompasses both malicious attacks and intrusions on a cloud provider and also 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].

The register abstraction provided by DEPSKY 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]. This semantics is both intuitive and stronger than the eventual consistency of some cloud-based services [Vog09]. Nevertheless, if the semantics provided by the underlying storage clouds is weaker than regular, then DEPSKY's semantics is also weaker (Section 6.3.10).

Notice that our model hides most of the complexity of the distributed storage system employed by the cloud provider to manage the data storage service since it just assumes that the service is an object storage system prone to Byzantine faults 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 an answer 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 an at most once semantics for the operations being invoked. This 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.

6.3.4 Protocol Design Rationale

Quorum protocols can serve as the backbone of highly available storage systems [CGKV09]. There are many protocols for implementing Byzantine fault-tolerant (BFT) storage [CT06, GWGR04, HGR07, LR06, MR98a, MR98b, MAD02], but most of them require that the servers execute some code, a functionality not available on storage clouds. This leads to a key difference between the DEPSKY protocols and these classical BFT protocols: *metadata and data are written in separate quorum accesses*. Moreover, 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.

There are also some quorum protocols that consider individual storage nodes as passive shared memory objects (or disks) instead of servers [ACKM06, ABO03, 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. 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 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, MR98b]. None of these mechanisms is implementable in general-purpose storage clouds.

To ensure confidentiality of stored data on the clouds without requiring a key distribution service, we employ a *secret sharing scheme* [Sha79]. In this scheme, a special party called dealer distributes a secret to n players, but each player gets only a share of this secret. The main properties of the scheme is that at least $f+1 \leq n$ 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 receives just a share of the data being written, besides the metadata. This ensures that no individual cloud will have access to the data being stored, but that clients that have authorization to access the data will be granted access to the shares of (at least) $f+1$ different clouds and will be able to rebuild the original 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.

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 to not erase 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.

6.3.5 DEPSKY-A– Available DepSky

The first DEPSKY protocol is called DEPSKY-A, and improves the availability and integrity of cloud-stored data by replicating it on several providers using quorum techniques. Algorithm 1 presents this protocol. Due to space constraints we encapsulate some of the protocol steps in the functions of the first two rows of Table 6.3.5. 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.

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 (lines 12). 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 does its first writing in a data unit *du* (lines 3-5, max_ver_{du} is initialized as 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* just fetches the metadata files from a quorum of clouds (line 16), chooses the one with greatest version number (line 17) and reads the value corresponding to this version number and the cryptographic hash found in the chosen metadata (lines 19-22). After receiving the first reply that satisfies this condition the reader cancels the pending RPCs and returns the value (lines 22-24).

The rationale of why this protocol provides the desired properties is the following (proofs on the Appendix). 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

Function	Description
$queryMetadata(du)$	obtains the correctly signed file metadata stored in the container du of $n-f$ out-of the n 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. Blocks until it receives write confirmations from $n-f$ clouds.
$\mathsf{H}(value)$	returns the cryptographic hash of $value$.

Table 6.1: Functions used in the DEPSKY-A protocols.

Algorithm 1: DEPSKY-A

```
procedure DepSkyAWrite(du,value)
begin
    if $max\_ver_{du} = 0$ then
        $m \longleftarrow queryMetadata(du)$
        $max\_ver_{du} \longleftarrow \max(\{m[i].ver \ : \ 0 \le i \le n-1\})$
    $new\_ver \longleftarrow max\_ver_{du} + 1$
    $v[0 \mathinner{..} n-1] \longleftarrow value$
    $writeQuorum(du, \text{“value”} + new\_ver, v)$
    $new\_meta \longleftarrow \langle new\_ver, \mathsf{H}(value) \rangle$
    $sign(new\_meta, K_{r_w})$
    $v[0 \mathinner{..} n-1] \longleftarrow new\_meta$
    $writeQuorum(du, \text{“metadata”}, v)$
    $max\_ver_{du} \longleftarrow new\_ver$

function DepSkyARead(du)
begin
    $m \longleftarrow queryMetadata(du)$
    $max\_id \longleftarrow i \ : \ m[i].ver = \max(\{m[i].ver \ : \ 0 \le i \le n-1\})$
    $v[0 \mathinner{..} n-1] \longleftarrow \perp$
    parallel for $0 \le i < n-1$ do
        $tmp_i \longleftarrow cloud_i.get(du, \text{“value”} + m[max\_id].ver)$
        if $\mathsf{H}(tmp_i) = m[max\_id].digest$ then $v[i] \longleftarrow tmp_i$
    wait until $\exists i : v[i] \neq \perp$
    for $0 \le i \le n-1$ do $cloud_i.cancel\_pending()$
    return $v[i]$
```

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.

6.3.6 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[1].

The DEPSKY-CA protocol integrates these techniques with the DEPSKY-A protocol (Algorithm 2). The additional cryptographic and coding functions needed are in Table 6.3.6.

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 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

Table 6.2: Functions used in the DEPSKY-CA protocols.

The DEPSKY-CA protocol is very similar to DEPSKY-A with the following differences: (1.) the encryption 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 30-31); (2.) the data stored in $cloud_i$ is composed by the share of the key $s[i]$ and the encoded block $e[i]$ (lines 12, 30-31); and (3.) $f+1$ replies are necessary to read the data unit's current value instead of one on DEPSKY-A (line 28). 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 31 are not necessary).

The rationale of the correctness of the protocol is similar to the one for DEPSKY-A. 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.

6.3.7 Read Optimization

The DEPSKY-A algorithm described in Section 6.3.5 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 it is far from ideal: even using just a single value, the application will be paying for n data accesses. A lower-cost solution is to use some criteria to sort the clouds and try to access them sequentially, one at time, until we obtain the desired value. The sorting criteria can be based on access monetary cost (cost-optimal), the latency of *queryMetadata* on the protocol

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

Algorithm 2: DEPSKY-CA

```
procedure DepSkyCAWrite(du,value)
begin
    if $max\_ver_{du} = 0$ then
        $m \longleftarrow queryMetadata(du)$
        $max\_ver_{du} \longleftarrow \max(\{m[i].version : 0 \leq i \leq n-1\})$
    $new\_ver \longleftarrow max\_ver_{du} + 1$
    $k \longleftarrow generateSecretKey()$
    $e \longleftarrow \mathsf{E}(value, k)$
    $s[0 .. n-1] \longleftarrow share(k, n, f+1)$
    $v[0 .. n-1] \longleftarrow encode(e, n, f+1)$
    for $0 \leq i < n-1$ do
        $d[i] \longleftarrow \langle s[i], e[i] \rangle$
        $h[i] \longleftarrow \mathsf{H}(d[i])$
    $writeQuorum(du,$"value"$+new\_ver, d)$
    $new\_meta \longleftarrow \langle new\_ver, h \rangle$
    $sign(new\_meta, K_{r_w})$
    $v[0 .. n-1] \longleftarrow new\_meta$
    $writeQuorum(du,$"metadata"$, v)$
    $max\_ver_{du} \longleftarrow new\_ver$

function DepSkyCARead(du)
begin
    $m \longleftarrow queryMetadata(du)$
    $max\_id \longleftarrow i : m[i].ver = \max(\{m[i].ver : 0 \leq i \leq n-1\})$
    $d[0 .. n-1] \longleftarrow \perp$
    parallel for $0 \leq i \leq n-1$ do
        $tmp_i \longleftarrow cloud_i.get(du,$ "value" $+m[max\_id].ver)$
        if $\mathsf{H}(tmp_i) = m[max\_id].digest[i]$ then $d[i] \longleftarrow tmp_i$
    wait until $|\{i : d[i] \neq \perp\}| > f$
    for $0 \leq i \leq n-1$ do $cloud_i.cancel\_pending()$
    $e \longleftarrow decode(d.e, n, f+1)$
    $k \longleftarrow combine(d.s, n, f+1)$
    return $\mathsf{D}(e, k)$
```

(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.

6.3.8 Supporting Multiple Writers – Locks

The DEPSKY protocols presented do not support concurrent writes, which is sufficient for many applications where each process write 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 backend 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, a coordination system like Zookeeper [HKJR10a] 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).

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. The protocol is the following:

1. A process p that wants to be a writer (and has permission to be), first lists files on the data unit container on all n clouds and tries to find a zero-byte file called *lock-ID-T*. If such file is found on a quorum of clouds, $ID \neq p$ and the local time t on the process is smaller than $T + \Delta$, being Δ a safety margin concerning the difference between the synchronized clocks of all writers, someone else is the writer and p will wait until $T + \Delta$.

2. If the test fails, p can write a lock file called *lock-p-T* on all clouds, being $T = t + writer\_lease\_time$.

3. In the last step p lists again all files in the data unit container searching for other lock files with $t < T + \Delta$ besides the one it wrote. If such file is found, p 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. Otherwise, p becomes the single-writer for the data unit until T.

Several remarks can be made about this protocol. First, the last step 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. Moreover, unlocking can be easily done through the removal of the lock files. Third, the protocol requires synchronized clocks in order to employ leases and thus tolerate writer crashes. Finally, 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 on step 3, this situation should be very rare on the envisioned deployments for the systems.

6.3.9 Additional Protocols

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 data units 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 6.3.4, 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 too costly for the organization and thus some garbage collection is necessary. The protocol for doing that is very simple: a writer just lists all files *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.

Cloud reconfiguration. Sometimes one cloud can become too expensive or too unreliable to be used for storing DEPSKY data units. In this case we need a reconfiguration protocol to move the blocks from one cloud to another. The process is the following: (1.) the writer reads the data (probably from the other clouds and not from the one being removed); (2.) it 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.

6.3.10 Dealing with Weakly Consistent Clouds

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

With a slight modification, our protocols can work with these weakly consistent clouds. The modification is very simple: repeat the data file reading from the clouds until the required condition is satisfied (receiving 1 or $f+1$ data units, respectively in lines 22 and 28 of Algorithms 1 and 2). This modification ensures the read of a value described on a read metadata will be repeated until it is available.

This modification makes the DEPSKY protocols be *consistency-proportional* in the following sense: if the underlaying clouds provide regular semantics, the protocols provide regular semantics; if the clouds provide read-after-write semantics, the protocol satisfies read-after-write semantics; and finally, if the clouds provide eventually consistency, the protocols are eventually consistent. Notice that if the underlying clouds are heterogeneous in terms of consistency guarantees, DEPSKY ensures the weakest consistency among those provided. This comes from the fact that reading of a recently write value depends on the reading of the new metadata file, which, after a write is complete, will only be available eventually on weakly consistent clouds.

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 an unique id of the writer (e.g., the hash of part of its private key) as the decimal part of its timestamps, just like is done in most Byzantine quorum protocols (e.g., [MR98a]). 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.

6.4 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 protocols; and (3) drivers to access cloud providers, which implement 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 2910 lines of code, being 1122 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 provider.

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 code, respectively.

6.5 Related Work

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 (which is substantially different than 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., [MR98a, GWGR04, HGR07, MAD02]), however, few of them address the model in which servers are passive entities that do not run protocol code [ACKM06, ABO03, 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] presents a single-writer multi-reader regular register construction that requires two communication round-trips both for reading and writing in absence of contention. 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 [ABO03] 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.

6.6 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. We believe DEPSKY protocols are in an unexplored region of quorum systems design space and can enable applications sharing critical data (e.g., financial, medical) to benefit from clouds.

Chapter 7

Consistency for untrusted service execution

Chapter Author:
Christian Cachin (IBM)

7.1 Introduction

As stated in previous chapters, ensuring the *integrity* of remote data and outsourced computations in the context of cloud computing is a very severe problem. Clients depend on the services they use. A bug in the service implementation or a malicious adversary who compromised the corresponding server(s) can have serious consequences. This gets even worse, when services are not executed within a local and controllable context, but by a third party, namely the cloud provider. Therefore, this work aims at discovering such misbehavior, in order for the clients to take some compensation action.

When the service provides data storage (read and write operations only), some well-known methods guarantee data integrity. With only one client, a *memory checker* [BEG+94] ensures that a read operation always returns the most recently written value. If multiple clients access the remote storage, they can combine a memory checker with an external trusted infrastructure (like a directory service or a key manager in a cryptographic file system), and achieve the same guarantees for many clients.

But in the asynchronous network model without client-to-client communication considered here, nothing prevents the server from mounting a *forking attack*, whereby it simply omits the operations of one client in its responses to other clients. Mazières and Shasha [MS02] put forward the notion of *fork-linearizability*, which captures the optimal achievable consistency guarantee in this setting. It ensures that whenever the server's responses to a client A have ignored a write operation executed by a client B, then A can never again read a value written by B afterwards and vice versa. With this notion, the clients detect server misbehavior from a single inconsistent operation — this is much easier than comparing the effects of *all* past operations one-by-one.

This work makes the first step toward ensuring integrity and consistency for *arbitrary computing services* running on an untrusted server. It does so by extending untrusted storage protocols providing fork-linearizability to a generic service protocol with fork-linearizable semantics. Previous work in this model only addressed integrity for a storage service, but could not check the consistency of more general computations by the server.

Similar to the case of a storage service, the server can readily mount a forking attack by splitting the group of clients into subgroups and responding consistently within each subgroup, but not making operations from one subgroup visible to others. Because the protocol presented here ensures fork-linearizability, however, such violations become easy to discover. The method therefore protects the integrity of arbitrary services in an end-to-end way, as opposed to existing techniques that aim at ensuring the integrity of a computing platform (e.g., the *trusted computing* paradigm).

Our approach requires that (at least part of) the service implementation is known to the clients, because they need to double-check crucial steps of an algorithm locally. In this sense, the notion of fork-linearizable service integrity, as considered here, means that the clients have collaboratively verified every single operation of the service. This strictly generalizes the established notion of fork-linearizable storage integrity. A related notion for databases is ensured by the Blind Stone Tablet protocol [WSS09].

7.1.1 Contributions

We present the first precise model for a group of mutually trusting clients to execute an *arbitrary service* on an untrusted server S, with the following characteristics. It guarantees *atomic operations* to all clients when S is correct and *fork-linearizability* when S is faulty; this means that all clients which observe each other's operations are *consistent*, in the sense that their own operations, plus those operations whose effects they see, have occurred atomically in same sequence.

Furthermore, we generalize the concept of *authenticated data structures* [NN00] toward executing arbitrary services in an authenticated manner with multiple clients. We present a protocol for consistent service execution on an untrusted server, which adds $O(n)$ communication overhead for a group of n clients; it generalizes existing protocols that have addressed only the special case of storage on an untrusted server.

7.1.2 Organization

Section 7.2 describes the model and recalls fork-linearizability and other consistency notions. In Section 7.3 the notion of *authenticated service execution* is introduced, which plays the main role for formalizing arbitrary services so that their responses can be verified. Section 7.4 presents the fork-linearizable service execution protocol. The detailed analysis and generalizations are omitted from this chapter.

7.2 System model

7.2.1 System

We consider an asynchronous distributed system consisting of n clients $C_1, \ldots, C_n$ and a server S. Every client is connected to S through an asynchronous reliable channel that delivers messages in first-in/first-out (FIFO) order. The clients and the server together are called *parties*. A *protocol P* specifies the behaviors of all parties. An *execution* of P is a sequence of alternating states and state transitions, called *events*, which occur according to the specification of the system components.

All clients follow the protocol; in particular, they do not crash. Every client has some small local trusted memory, which serves to store keys and authentication values. The server might be faulty and deviate arbitrarily from the protocol; such behavior is also called *Byzantine*. A party that does not fail in an execution is *correct*.

7.2.2 Functionality

We consider a *deterministic state machine*, which is modeled by a *functionality* F as follows. It maintains a *state* $s \in \mathscr{S}$, repeatedly takes some *operation* $o \in \mathscr{O}$ as input (o may contain arguments), and outputs a *response* $r \in \mathscr{R}$ and a new state s'. The initial state is denoted by s_{F0}. Formally, a step of F is written as

$$(s', r) \leftarrow F(s, o).$$

Because operations are executed one after another, this gives the *sequential specification* of F. We discuss the concurrent invocation of multiple operations later.

We extend this notation for executing multiple operations $o_1, \ldots, o_m$ in sequence, starting from an initial state s_0, and write

$$(s', r) = F(s_0, [o_1, \ldots, o_m])$$

for $(s_i, r_i) = F(s_{i-1}, o_i)$ with $i = 1, \ldots, m$ and $(s', r) = (s_m, r_m)$.

We define the *space complexity* of F, denoted by $SPACE_F$, to be the number of bits required to store the largest of its states, i.e.,

$$SPACE_F = \max_{s \in \mathscr{S}} |s|.$$

The space complexity determines the amount of local storage necessary to execute F.

7.2.3 Operations and histories

Our goal is to emulate F to the clients with the help of server S. The clients invoke the operations of F; every operation is represented by two events occurring at the client, an *invocation* and a *response*. A *history* of an execution σ consists of the sequence of invocations and responses of F occurring in σ. An operation is *complete* in a history if it has a matching response. For a sequence of events σ, *complete*(σ) is the maximal subsequence of σ consisting only of complete operations.

An operation o *precedes* another operation o' in a sequence of events σ, denoted $o <_\sigma o'$, whenever o completes before o' is invoked in σ. A sequence of events π *preserves the real-time order* of a history σ if for every two operations o and o' in π, if $o <_\sigma o'$ then $o <_\pi o'$. Two operations are *concurrent* if neither one of them precedes the other. A sequence of events is *sequential* if it does not contain concurrent operations. For a sequence of events σ, the subsequence of σ consisting only of events occurring at client C_i is denoted by $\sigma|_{C_i}$ (we use the symbol $|$ as a projection operator). For some operation o, the prefix of σ that ends with the last event of o is denoted by $\sigma|^o$.

An execution is *well-formed* if the sequence of events at each client consists of alternating invocations and matching responses, starting with an invocation. An execution is *fair*, informally, if it does not halt prematurely when there are still steps to be taken or messages to be delivered.

7.2.4 Consistency conditions

We now describe the formal consistency notions required from an untrusted service, formulated in terms of the possible views of a client. A sequence of events π is called a *view* of a history σ at a client C_i w.r.t. a functionality F if σ can be extended (by appending zero or more responses) to a history σ' such that:

1. π is a sequential permutation of some subsequence of $complete(\sigma')$;

2. $\pi|_{C_i} = complete(\sigma')|_{C_i}$; and

3. π satisfies the sequential specification of F.

Intuitively, a view π of σ at C_i contains at least all those operations that either occur at C_i or are apparent from to C_i from its interaction with F.

One of the most important consistency conditions for concurrent operations is linearizability, which guarantees that all operations occur atomically.

Definition 1 (Linearizability [HW90b]). A history σ is *linearizable* w.r.t. a functionality F if there exists a sequence of events π such that:

1. π is a view of σ at all clients w.r.t. F; and

2. π preserves the real-time order of σ.

The notion of fork-linearizability [MS02] (originally called *fork consistency*) requires that when an operation is observed by multiple clients, the history of events occurring before the operation is the same. For instance, when a client reads a value written by another client from a storage service, the reader is assured to be consistent with the writer up to the write operation.

Definition 2 (Fork-linearizability). A history σ is *fork-linearizable* w.r.t. a functionality F if for each client C_i there exists a sequence of events π_i such that:

1. π_i is a view of σ at C_i w.r.t. F;

2. π_i preserves the real-time order of σ;

3. (*No-join*) For every client C_j and every operation $o \in \pi_i \cap \pi_j$, it holds that $\pi_i|^o = \pi_j|^o$.

We now recall the concept of a *fork-linearizable Byzantine emulation* [CSS07]. It summarizes the requirements put on our service emulation protocol, which runs between the clients and an untrusted server. This notion means that when the server is correct, the service should guarantee the standard notion of linearizability; otherwise, it should ensure fork-linearizability.

Definition 3 (Fork-linearizable Byzantine emulation). A protocol P *emulates* a functionality F *on a Byzantine server* S *with fork-linearizability* whenever the following conditions hold:

1. If S is correct, the history of every fair and well-formed execution of P is linearizable w.r.t. F; and

2. The history of every fair and well-formed execution of P is fork-linearizable w.r.t. F.

7.2.5 Cryptographic primitives

Our implementation uses *hash functions*, *digital signatures*, and *symmetric-key encryption*. We model them as ideal functionalities here. But all notions can be made formal in the model of modern cryptography.

A hash function H maps a bit string x of arbitrary length to a short, unique representation of fixed length. It is assumed to be collision-free, that is, no party can produce two different inputs x and x' such that $H(x) = H(x')$.

A digital signature scheme provides two operations, *sign* and *verify*. The invocation of *sign* takes an index $i \in \{1,\dots,n\}$ and a bit string m as parameters and returns a signature ϕ with the response. The *verify* operation takes the index i of a client, a string m, and a putative signature ϕ as parameters and returns a Boolean value $b \in \{\text{FALSE},\text{TRUE}\}$ with the response. It satisfies that $verify(i,m,\phi) = \text{TRUE}$ for all i and m if and only if C_i has executed $sign(i,m) = \phi$ before. Only C_i may invoke $sign(i,\cdot)$ and S cannot invoke *sign*. Every party may invoke *verify*.

A symmetric encryption scheme consists of a key generation algorithm, an encryption algorithm *encrypt* and a decryption algorithm *decrypt*. Initially a trusted entity runs the key generator and obtains a key $k \in \mathcal{K}$. Algorithm *encrypt* takes k and a message m as inputs and returns a ciphertext c. Algorithm *decrypt* takes k and a ciphertext c as inputs and returns a message m. For any k and m, it is required that $decrypt(k, encrypt(k,m)) = m$. Furthermore, any party that obtains $c = encrypt(k,m)$ but has no access to k obtains no useful information about m.

7.3 Service execution and authentication

This section first introduces a model for executing the service F on server S such that operations are invoked by the clients. The primary task of S is to maintain the global state s of F; we intend this model for coordination services, shared collaboration spaces, light-weight databases, storage applications and so on, with small computational expense for every operation, but high demand on maintaining a consistent state.

Given this setting, the clients could simply send their operations to S and, since F is deterministic, S could execute them and return the responses. But we are interested in a model where the clients execute the bulk of every operation, so as to reduce the load on S. This assumption also helps preparing the ground for authenticating the responses of S.

In the second part of this section, we introduce a model for *authenticating* the execution of a sequence of operations issued by a single client (imagine for a moment there is only one client; we extend this to multiple clients later). The client uses its local trusted memory to maintain some authentication data, from which it verifies the responses of F sent by S. This model closely resembles the established concept of authenticated data structures.

7.3.1 Separated execution

We model the execution of operations of F in a *separated way*, such that the clients do most of the work. Not all functionalities encountered require that every operation accesses the complete state s. An operation o can be executed in a separated way when it uses only a part s_o of the *global state* s of the functionality; this part may depend on the operation. If o modifies the global state, then the separated execution will also generate an updated state s'_o, which must be reconciled with s to maintain the correct semantics of F.

More formally, we say a functionality F allows *separated execution* when there exist three deterministic algorithms $extract_F$, $exec_F$, and $reconcile_F$ as follows. Algorithm $extract_F$ produces a *partial state* s_o from a global state s and an operation o,

$$s_o \leftarrow extract_F(s,o);$$

algorithm $exec_F$ executes o on the partial state s_o to produce a response r and a *partial updated state* s'_o,

$$(s'_o, r) \leftarrow exec_F(s_o, o);$$

finally, algorithm *reconcile$_F$* takes s'_o and o, together with the old global state s and outputs the new global state

$$s \leftarrow \mathit{reconcile}_F(s, s'_o, o).$$

The algorithms satisfy that for any $s \in \mathscr{S}$ and $o \in \mathscr{O}$, and for any s', r with $(s', r) = F(s, o)$, there exists a partial state $s_o = \mathit{extract}_F(s, o)$ and a partial updated state s'_o such that

$$(s'_o, r) = \mathit{exec}_F(s_o, o) \;\wedge\; s' = \mathit{reconcile}_F(s, s'_o, o)$$

and

$$|s_o| \ll |s| \;\wedge\; |s'_o| \ll |s'|.$$

In other words, the algorithms for the separated execution of F produce the same response and new state as the original F, but there exist intermediate states for the operation (s_o and s'_o), which are much smaller than the full state(s). The latter requirement should be understood qualitatively and is not quantified; but it is crucial for enabling efficient separated execution between a client and a server.

The *communication complexity* of some F with separated execution measures the size of the messages that must be communicated for separated execution. It is denoted by COMM_F and defined as the number of bits required to store the largest partial state s_o, partial updated state s'_o, together with a description of the operation o itself, for executing any operation on any state. That is,

$$\mathit{COMM}_F \;=\; \max\bigl\{|s_o| + |s'_o| + |o| \,\big|\, s \in \mathscr{S}, o \in \mathscr{O}, s_o = \mathit{extract}_F(s, o), (s'_o, r) = \mathit{exec}_F(s_o, o)\bigr\}.$$

7.3.2 Authenticated separated execution

When only a single client engages in separated execution of operations on the server, well-known methods allow the client to verify the correctness of the responses. These methods protect the client from a faulty server that tries to forge wrong responses. Known generally as *authenticated data structures* [NN00, MND+04], they apply to a broad class of information retrieval services, such as reading an item from a memory, hash tables, or search queries to a structured data type. Such service authentication schemes rely on a small *authenticator* value maintained by the client in its local trusted memory. The client can verify the response of an operation o in such a way that it recognizes when the response differs from the correct response r, resulting from applying o to the current state s of the service. That is, state s is obtained by applying all past operations of the client to F in order and the correct response is determined by $(s', r) = F(s, o)$. We model this concept as an extension of separated execution.

We say a functionality F allows *authenticated separated execution* when there exist three deterministic algorithms *authextract$_F$*, *authexec$_F$*, and *authreconcile$_F$* as follows. Algorithm *authextract$_F$* produces a *partial state s_o* from a global state s and an operation o,

$$s_o \leftarrow \mathit{authextract}_F(s, o).$$

The client maintains an *authenticator* denoted by a, which is initialized to a default value a_{F0}. Algorithm *authexec$_F$* takes a, s_o, and o as inputs and produces an *updated authenticator a'*, a *partial updated state s'_o*, and a response r. In the course of executing o, the algorithm also *verifies* its inputs with respect to a and may output the special symbol $\perp$ as response, indicating that the verification failed. In other words,

$$(a', s'_o, r) \leftarrow \mathit{authexec}_F(a, s_o, o),$$

with $r = \perp$ if and only if verification failed. Finally, algorithm $authreconcile_F$ takes s'_o and o, together with the old global state s and outputs the new global state

$$s \leftarrow authreconcile_F(s, s'_o, o).$$

Its role is exactly the same as in separated execution.

A *proper authenticated execution* of the operation sequence $o_1, \ldots, o_m$ proceeds as follows. Starting with the initial authenticator $a_0 = a_{F0}$ and state $s_0 = s_{F0}$, it computes

$$\begin{aligned} (s_i, r_i) &\leftarrow F(s_{i-1}, o_i) \\ s_{o_i} &\leftarrow authextract_F(s_i, o) \\ (a_i, s'_{o_i}, r_i) &\leftarrow authexec_F(a_{i-1}, s_{o_i}, o_i), \end{aligned}$$

for $i = 1, \ldots, m$ and outputs the triple (a_m, s_m, r_m) containing an authenticator a_m, state s_m, and response r_m.

Consider now the proper authenticated execution of an arbitrary operation sequence and the resulting authenticator a and state s. The following conditions must hold:

Correctness: For any $o \in \mathcal{O}$ and $(s', r) = F(s, o)$, there exist $s_o = authextract_F(s, o)$ and a', s'_o, and $r \neq \perp$ such that

$$(a', s'_o, r) = authexec_F(a, s_o, o) \land s' = authreconcile_F(s, s'_o, o).$$

and

$$|a'| \ll |s| \land |s_o| \ll |s| \land |s'_o| \ll |s'|.$$

Security: For any $o \in \mathcal{O}$ and any adversary that outputs some $\tilde{s}_o$, suppose that there exist a' and s'_o such that $(a', s'_o, \tilde{r}) = authexec_F(a, \tilde{s}_o, o)$ with $\tilde{r} \neq \perp$; then $\tilde{r} = r$.

The *correctness* property is simply reformulated from the unauthenticated scheme for separated execution. It states that for any authenticator and state s resulting from a proper authenticated execution, applying separated execution of o yields a response $r \neq \perp$ such that verification succeeds and, moreover, the resulting updated state s' together with r satisfies $(s', r) = F(s, o)$.

The *security* property considers a faulty S as an adversary, which tries to forge some partial state $\tilde{s}_o$ that causes the client to produce a wrong response $\tilde{r}$. But in an authenticated separated execution scheme, algorithm $authexec_F$ either outputs the correct response ($\tilde{r} = r$), or it recognizes the forgery and the verification fails ($\tilde{r} = \perp$).

The *communication complexity* of some F with authenticated separated execution is defined in the same way as for separated execution and measures how much data must be communicated between C and S.

The notion of authenticated data structures [MND+04] differs from a service with authenticated separated execution in that the former does not contain a partial updated state and the reconciliation step. In fact, the server could equally well execute the whole operation on the state that it maintains. But in practice, many algorithms execute update operations more efficiently when the client computes the updated parts of the state and the server merely stores them in its memory.

7.3.3 Examples

The literature contains many examples of data structures that can be formulated as functionalities with authenticated separated execution. They are interesting because their communication complexity for separated execution is much smaller than their space complexity. For instance, hash trees can be used to check the correctness of individual entries in a memory with N elements [BEG$^+$94] with complexity $O(\log N)$, a generalization of hash trees can authenticate responses produced by any DAG-structured query evaluation algorithm with logarithmic overhead [MND$^+$04], and cryptographic methods based on accumulators can maintain authenticated hash tables with constant communication for query operations and sub-linear cost for updates [PTT08].

As a concrete example, consider a functionality MEM whose state consists of N storage locations denoted by $MEM[1],\ldots,MEM[N]$. MEM supports two operations: $read(j)$, which returns $MEM[j]$, and $write((j,x))$, which assigns $MEM[j] \leftarrow x$ and returns nothing. Note that for $N = n$ and when C_i may only write to $MEM[i]$, we obtain the functionality that was considered in most previous work on untrusted storage (e.g., [CSS07]).

A standard hash tree computed over $MEM[1],\ldots,MEM[N]$ gives an authenticated separated execution scheme, where the internal nodes of the tree are also stored in the state of MEM. The authenticator is the root node of the hash tree, which commits all entries in MEM. Algorithm $authextract_{MEM}$ for an operation that concerns entry j always returns the internal tree nodes along the path from the root to the leaf node j and all their siblings, which are needed for recomputing the root hash in order to authenticate leaf node j [BEG$^+$94]. Verification succeeds if the recomputed root hash matches the authenticator. For a write operation, the nodes on the path from $MEM[j]$ to the root are updated and included in the partial updated state s'_o. The server extracts them from s'_o and stores them in the appropriate place during $authreconcile_{MEM}$.

The client must explicitly recompute the path in the hash tree also for write operations, in order to verify the sibling nodes along the path from the modified leaf node to the root; these nodes originate from the server and influence the computation of the new root hash. If they are not verified, they might lead to an invalid authenticator. Because the client computes these values anyway, they are contained in the partial updated state, and the server only needs to store them.

In this way, our notion of authenticated separated execution closely models what happens in practical hash tree implementations inside cryptographic storage systems; this is not possible with the notion of an authenticated data structure, where no reconciliation algorithm is foreseen.

7.4 Fork-linearizable execution protocol

We now introduce a novel untrusted service execution protocol, which emulates an arbitrary F on a Byzantine server with fork-linearizability. The protocol combines elements from existing untrusted storage protocols with an authenticated separated execution scheme for F.

The protocol operates in lock-step mode, similar to the bare-bones storage protocol of SUNDR [MS02]. This means that the server serializes all operations and does not allow them to execute concurrently. Proceeding in lock-step is for illustration purposes only; extending it to concurrent operations is feasible and discussed later.

At a high level, the protocol operates like this. A client assigns a local *timestamp* to every one of its operations. Every client maintains a timestamp vector T in its trusted memory. At client C_i, entry $T[j]$ is equal to the timestamp of the most recently executed operation by C_j in some view of C_i. To begin executing an operation o, client C_i sends a SUBMIT message with o to S. A correct S responds to this SUBMIT message by invoking the authenticated separated execution scheme, and computes $s_o \leftarrow \mathit{authextract}_F(s,o)$ on the current state s.

In addition to s, the server maintains a timestamp vector V, an authenticator a, and a signature φ, which it received in a so-called COMMIT message from the client C_c that executed the last preceding operation at S. The signature was issued by C_c on V and a. The server sends a REPLY message to C_i containing V, a, s_o, c, and φ.

When it receives the REPLY message, the client first checks the content. It verifies the signature φ and makes sure that $V \geq T$ (using vector comparison) and that $V[i] = T[i]$. If not, the client aborts the operation and halts, because this means that S has violated the consistency of the service.

Then C_i verifies the response with respect to a and runs the separated execution by computing $(a', s'_o, r) \leftarrow \mathit{authexec}_F(a, s_o, o)$. If the verification fails, the client again halts. Otherwise, C_i proceeds to copying the received timestamp vector V into its variable T, incrementing $T[i]$, and computing a signature φ' on T and a'. The value $T[i]$ becomes the *timestamp* of o. Finally, C_i returns a COMMIT message to S containing T, a', s'_o, and φ'.

It is not hard to see that all checks are satisfied when S is correct because every client only increments its own entry in a timestamp vector. Therefore, the timestamp vectors sent out by S in REPLY messages appear in strictly increasing order.

The description so far allows the server to learn the authenticator values, which is not foreseen in the model of an authenticated separated execution scheme. To prevent any damage that might be caused by this, all clients know a common secret key k for a symmetric encryption scheme and use it to encrypt the authenticator before sending it to S.

This completes the high-level description of the untrusted service execution protocol; the details are given in Algorithms 1 and 2.

Intuitively, the algorithm relies on the same properties of vector clocks as previous protocols for untrusted storage [MS02, CSS07]. Note that S can only send a timestamp vector and authenticator in a REPLY message that have been signed by a client; otherwise, the first verification step in Algorithm 1 fails. Under this condition, S may violate the protocol only by sending a timestamp vector/authenticator pair that is properly signed but does not satisfy a global sequential order of the operations.

In other words, a violation by S means that there is one operation o_0 whose timestamp vector is received in a REPLY by at least two different clients C_1 and C_2, in operations o_1 and o_2, respectively. If all other information is correct, operations o_1 and o_2 both succeed, but the two clients sign *incomparable* timestamp vectors. According to the protocol, one can then show that C_1 will not execute any operation in a view at C_1 that includes o_2 and, vice versa, any operation in a view at C_2 that includes o_1 will cause C_2 to abort.

With the functionality *MEM* from the previous section and n storage locations, this protocol gives the same guarantees as the bare-bones storage protocol of SUNDR [MS02] and the lock-step protocol of Cachin et al. [CSS07]. As in the latter protocol, our algorithm adds a linear (in n) overhead to the communication complexity of separated execution.

7.5 Conclusion

This chapter has introduced the first precise model for a group of mutually trusting clients to execute an arbitrary service on an untrusted server S, such that the clients observe atomic operations when S is correct and the service respects fork-linearizability when S is Byzantine.

Algorithm 1 Untrusted execution protocol for client C_i

```
State
    $k \in \mathcal{K}$                                              // symmetric encryption key
    $T \in \mathbb{N}_0{}^n$, initially $[0]^n$                       // current timestamp vector

upon operation $run_F(o)$ do
    send message $[\text{SUBMIT}, o]$ to $S$
    wait for message $[\text{REPLY}, V, \bar{a}, s_o, c, \varphi]$
    if $\big(V = [0]^n \vee verify(c, \text{COMMIT}\|V\|\bar{a}, \varphi)\big) \wedge V \geq T \wedge V[i] = T[i]$ then
        if $V = [0]^n$ then
            $a \leftarrow a_{F0}$
        else
            $a \leftarrow decrypt(k, \bar{a})$
        $(a', s'_o, r) \leftarrow authexec_F(a, s_o, o)$
        if $r \neq \perp$ then
            $T \leftarrow V$
            $T[i] \leftarrow T[i] + 1$
            $\varphi' \leftarrow sign(i, \text{COMMIT}\|T\|a')$
            $\bar{a}' \leftarrow encrypt(k, a')$
            send message $[\text{COMMIT}, o, T, \bar{a}', s'_o, \varphi']$ to $S$
            return $r$
    halt
```

Algorithm 2 Untrusted execution protocol for server S

```
State
    $s \in \mathcal{S}$, initially $s_{F0}$                                        // state of $F$
    $c \in \{1, \ldots, n\}$, initially 1          // index of currently or most recently served client
    $V \in \mathbb{N}_0{}^n$, initially $[0]^n$             // timestamp vector of last committed operation
    $\bar{a}$, initially $\varepsilon$                    // encrypted authenticator of last committed operation
    $\varphi$, initially $\varepsilon$                                  // signature of last committed operation
    $block \in \{\text{FALSE}, \text{TRUE}\}$, initially FALSE

upon receiving message $[\text{SUBMIT}, o]$ from $C_i$ such that $block = \text{FALSE}$ do
    $s_o \leftarrow authextract_F(s, o)$
    send message $[\text{REPLY}, V, \bar{a}, s_o, c, \varphi]$ to $C_i$
    $c \leftarrow i$
    $block \leftarrow \text{TRUE}$

upon receiving message $[\text{COMMIT}, o, T, \bar{a}', s'_o, \varphi']$ from $C_i$ such that $block = \text{TRUE} \wedge i = c$ do
    $s \leftarrow authreconcile_F(s, s'_o, o)$
    $(V, \bar{a}, \varphi) \leftarrow (T, \bar{a}', \varphi')$
    $block \leftarrow \text{FALSE}$
```

An implementation of this notion has been obtained by combining any scheme for authenticated separated execution with elements from untrusted storage protocols.

The protocol is not particularly efficient because a correct server executes all operations in lock-step mode. Similar to untrusted storage protocols, the protocol can be improved by letting the clients execute some operations concurrently, as long as they do not conflict. Some restrictions on the achievable parallelism have been identified [CSS07].

Chapter 8

State Machine Replication

Chapter Authors:
Alysson Bessani (FFCUL) and JoÃčo Sousa (FFCUL)

8.1 Introduction

The last decade saw an impressive amount of papers on Byzantine Fault-Tolerant (BFT) State Machine Replication (SMR) (e.g., [CL02, CWA$^+$09a, VCBL09, CKL$^+$09], to cite just a few), but almost no practical use of these techniques in real deployments.

Our view of this situation is that the fact that there is no robust-enough implementation of BFT SMR available makes it quite difficult to use this kind of technique, since implementing this type of protocol is very far from trivial. To the best of our knowledge, from all "BFT systems" that appeared on the last decade, only the original PBFT system [CL02] and the very recent UpRight [CKL$^+$09] implement a complete replication system (which deal with the normal synchronous fault-free case and all corner cases that happen when there are faults and asynchrony). However, our experience with PBFT shows that it is not robust enough (e.g., we could not make it survive a primary failure) and it is not being maintained anymore, and UpRight uses a 3-tier architecture which tends to be more than a simple BFT replication library.

In this chapter we describe a 3-year effort in implementing BFT-SMaRt [sma], a Java-based BFT SMR library which implements a protocol similar to the one used in PBFT but targets not only high-performance in fault-free executions, but also correctness in all possible malicious behaviors of faulty replicas.

The main contribution of this chapter is to fill a gap in the BFT literature by documenting the implementation of this kind of protocol, and not the protocol itself.

The chapter is organized in the following way: Section 8.2 enumerates the design principles of BFT-SMaRt. Section 8.3 briefly describes the replication protocol implemented by BFT-SMaRt. The replica architecture and its modules are presented on Section 8.4. Sections 8.5 and 8.6 present a brief overview of how to use the library to implement dependable services and what kind of performance numbers can be expected from this library. Finally, Sections 8.7 and 8.8 discuss some lessons learned on BFT-SMaRt implementation and our concluding remarks.

8.2 BFT-SMaRt Design Principles

The development of BFT-SMaRt started at the beginning of 2007 aiming to build a BFT total order multicast library for the replication layer of the DepSpace coordination service [BACF08].

This first version was called JBP (Java Byzantine Paxos) and is still available on the DepSpace web site [jit]. In the last year we revamped the design of this multicast library to make it a complete BFT replication library, including features such as checkpoints and state transfer. Since the beginning, BFT-SMaRt was developed with the following design principles in mind:

- *Java:* for security, portability, ease of programming and maintenance, we choose the Java programming language and opted to meet the challenge of making a high performance BFT implementation using a programming language believed to be much less performant than C (which is used in other BFT implementations);
- *Modularity:* most high-performance BFT state machine replication algorithms (e.g., [CL02, VCBL09]) are described as a monolithic software implementation in which a set of mechanisms such as Paxos consensus, total order multicast, checkpointing, state transfer, client management, leader election plus possible optimizations were integrated on a protocol that ensures the linearizability of the replicated service. BFT-SMaRt, on the other hand, was designed taking into account several decoupled modules, being the most notable the separation between Byzantine Paxos consensus, total order multicast, and checkpointing/state transfer;
- *No optimizations that bring complexity:* a recent paper [CWA+09a] showed that the use of fragile optimizations can make a BFT protocol more susceptible to performance degradation attacks. One principle of BFT-SMaRt is not to use many optimizations proposed on past works in order to avoid the code complexity of using them under an asynchronous system. Consequently, the current version of our library does not implement many optimizations commonly advocated on BFT papers (e.g., agreement over hashes, speculation) and still provides decent performance numbers.

These design principles are pragmatic and reflect our final objective with BFT-SMaRt: providing a real-world complete implementation of a replication library that could be used and evolved by the research community, and not to have a prototype for showing that a particular protocol can be implemented.

8.3 The Replication Protocol

In this section we give a brief overview of how we used a Byzantine Paxos consensus protocol called Paxos at War (PaW) [Zie04] to implement state machine replication on BFT-SMaRt. The message pattern is presented in Figure 8.1 and is very similar to the well known PBFT [CL02].

Underlying assumptions Our protocol assumes a set of $n \geq 3f+1$ replicas and an unknown number of clients. Up to f replicas and an unbounded number of clients can fail arbitrarily. We assume an eventually synchronous system model like other protocols for BFT SMR [CL02]. We assume also reliable authenticated point-to-point links between processes. These links are implemented using message authentication codes (MACs) over TCP/IP, which requires a public-key infrastructure to ensure that each process can receive and validate the public key of every other process (which is a requirement in order to use Signed Diffie-Hellman to generate the shared keys used for MACs).

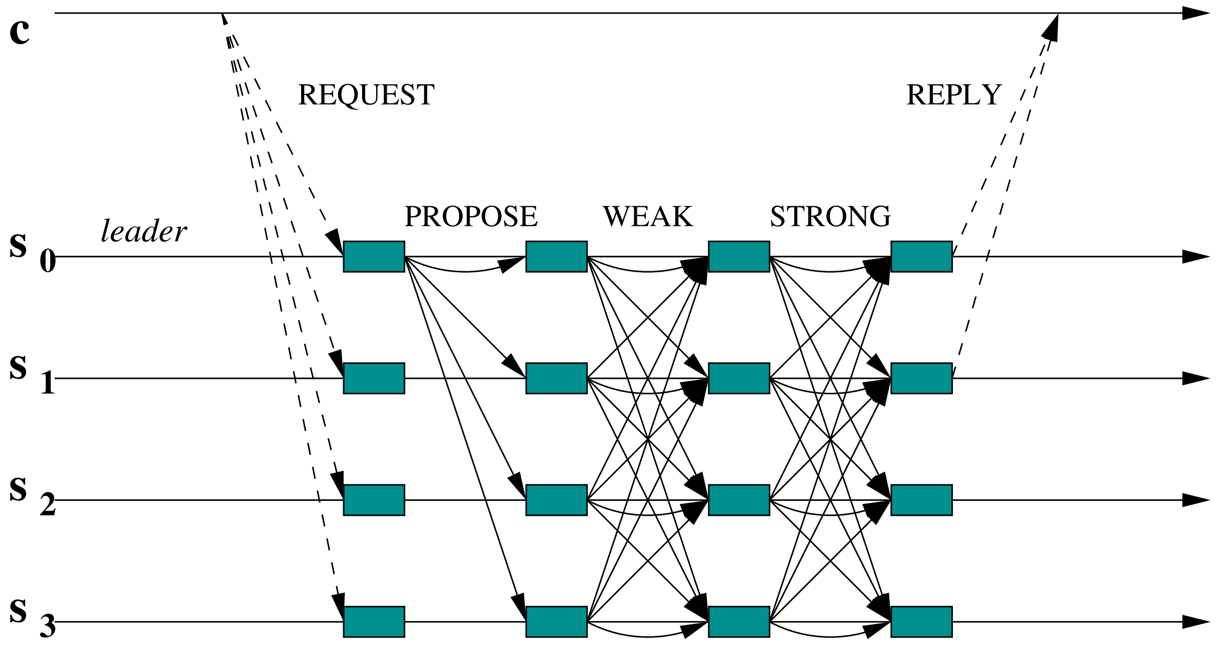


Figure 8.1: The message pattern of a BFT-SMaRt fault-free execution.

Paxos at War A consensus execution i begins with one of the replicas designated as the leader (initially the replica with the lowest id) proposing some value for the consensus through a PROPOSE message. All replicas that receive this message, verify if its sender is the current leader, and if the value proposed is good[1], they weakly accept the value being proposed, sending a WEAK message to all other replicas. If some replica receives more than $\frac{n+f}{2}$ WEAK accepts for the same value, it strongly accepts this value and sends a STRONG message to all other replicas. If some replica receives more than $\frac{n+f}{2}$ STRONG accepts for the same value, this value is used as the decision for consensus. If some replica stays on the same round of a consensus execution for more than a pre-defined timeout it should freeze the round and send a FREEZE message to all other replicas. When a round is frozen the replica does not accept PROPOSE, WEAK or STRONG messages for this round. All replicas that receive more than f FREEZE messages should freeze its round (if it is not already frozen) and send a signed COLLECT message to the leader of the next round[2]. This message specifies the state of the replica and allows the leader of the next round to choose a value to be proposed that does not contradict previous decided values.

From consensus to total order multicast It appears to be reasonably simple to build a total order multicast primitive using a Paxos consensus protocol: the decision value of the consensus instance number i is the i-th set of messages to be delivered by the total order multicast [MA06]. However, there are some problems overlooked by this transformation. First, a leader can propose any value for consensus, and we have to develop some mechanism to ensure that forged messages (i.e., not sent by clients) are not accepted as good values. Second, we have to deal with the case in which a malicious leader always follows the consensus protocol but does not

[1]The proposal of the first consensus round is always good, in other rounds some proof of goodness should be sent together with the value [Zie04].

[2]All replicas know what will be the leader of the next round given that it is calculated in a deterministic way. See [Zie04] for more details.

propose messages from certain clients. The first problem can be easily solved by making clients sign their messages using public-key signatures or authenticators (MAC vectors) [CL02]. The second problem can be addressed with timers associated to (pending) messages received from clients. In this way, non-leader replicas monitor if the leader proposes the ordering of each message within a certain time bound. The complexity of this solution is what to do when the the timer expires. Our solution is to freeze the current round of the consensus being executed and force a leader change in a similar way to what is defined by PaW.

Logging, checkpoints and state transfer To implement a practical state machine replication, the replicas should be able to be repaired and reintegrated on the system, without restarting the whole replicated service. This can only be done if we provide a state transfer protocol between the replicas. The idea is similar to what is used on [CL02] (and many others): when a replica discovers that it is too late in relation to other replicas (e.g., because it is using a slow network link), it triggers a state transfer operation. This operation consists in the recovering replica sending a STATE-REQUEST message to the other replicas asking for the state containing the execution of the messages decided on consensus $0...u-1$ (being u the first consensus decision that the replica knows about after the recovery) and waiting for $f+1$ STATE replies with the same state, which is used to update the service state. For the replicas to be able to answer the STATE-REQUEST, they should log the operations executed for each consensus (i.e., the decision value) and collect checkpoints periodically (after each c consensus executions) to trunk this log.

8.4 BFT-SMaRt Replica Architecture

8.4.1 Building blocks

To achieve the modularity goal, we defined a set of building blocks (or modules) that should contain the core of the functionalities required by BFT-SMaRt. These blocks are divided in two groups: *communication system* and *state machine replication*. The latter implements the BFT-SMaRt replication protocol described in Section 8.3, while the former encapsulates everything related to client-to-replica and replica-to-replica communication, including authentication, replay attacks detection, and reestablishment of communication channels after a network failure.

Communication system

The communication system provides a queue abstraction for receiving both requests from clients and messages from other replicas, as well as a simple send method that allows a replica to send a byte array to some other replica or client identified by an integer. There are three main modules here:

- **Client Communication System**: this module deals with the clients that connect, send requests and receive responses from replicas. Given the open-nature of this communication (since replicas can serve an unbounded number of clients) we choose the Netty communication framework [net] for implementing client/server communication. The most important requirement of this module is that it should be able to accept and deal with hundreds of connections efficiently. To do this, the Netty framework uses the `java.nio.Selector` class and a configurable thread pool.

- **Client Manager**: after receiving a request from a client, this request should be verified and stored to be used by the replication protocol. For each known client, this module stores the sequence number of the last request received from this client (to detect replay attacks) and maintains a queue containing the requests received but not yet delivered to the service being replicated (that we call service replica). The requests to be ordered in a consensus are taken from these queues in a fair way.

- **Server Communication System**: while the replicas should accept connections from an unlimited number of clients, as is supported by the client communication system described above, the server communication system implements a closed-group communication model used by the replicas to send messages between themselves. The implementation of this layer was made through "usual" Java sockets, using one thread to send and one thread to receive for each server. One of the key responsibilities of this module is to reestablish the channels between every two replicas after a failure and a recovery.

State machine replication

Using the simple interface provided by the communication system to access reliable and authenticated point-to-point links, we have implemented the state machine replication protocol. BFT-SMART uses five main modules to achieve state machine replication.

- **Proposer**: this reasonable simple module (which contains a single class) implements the role of a proposer, i.e., how it can propose a value to be accepted and what a replica should do when it is elected as a new leader.

- **Acceptor**: this module implements the core of the PaW algorithm: messages of type WEAK and STRONG are processed and generated here. For instance, when more than $\frac{n+f}{2}$ WEAK messages for the same (proposed) value are received, a corresponding STRONG message is generated. Most of the complex code to deal with leader changes is also in this module.

- **Total Order Multicast (TOM)**: this module gets pending messages received by the client communication system and calls the proposer module to start a consensus instance. Additionally, a class of this module is responsible for delivering requests to the service replica and to create and destroy timers for the pending messages of each client.

- **Execution Manager**: this module is closely related to the TOM and is used to manage the execution of consensus instances. It stores information about consensus instances and their rounds as well as who was the leader replica on these rounds. Moreover, the execution manager is responsible to stop and re-start a consensus being executed (to force leader changes, as described in Section 8.3).

- **State Manager**: checkpointing and state transfer is implemented by this module, which stores in a state log both the current checkpoint and the messages delivered since it was taken. When a new checkpoint is done, the previous content of the state log is deleted. This log is used to answer STATE-REQUEST messages received from other replicas.

8.4.2 Staged Message Processing

A key point when implementing a high-throughput replication protocol is how to break the several tasks of the protocol in a suitable architecture that can be robust and efficient. In the case of BFT SMR there are two additional requirements: the system should deal with hundreds of clients and resist malicious behaviours from both replicas and clients.

Figure 8.2 presents the main architecture with the threads used for staged message processing of the protocol implementation. In this architecture, all threads communicate through *bounded queues* and the figure shows which thread feeds and consumes data from which queues. This architecture was heavily influenced by the SEDA framework [WCB01].

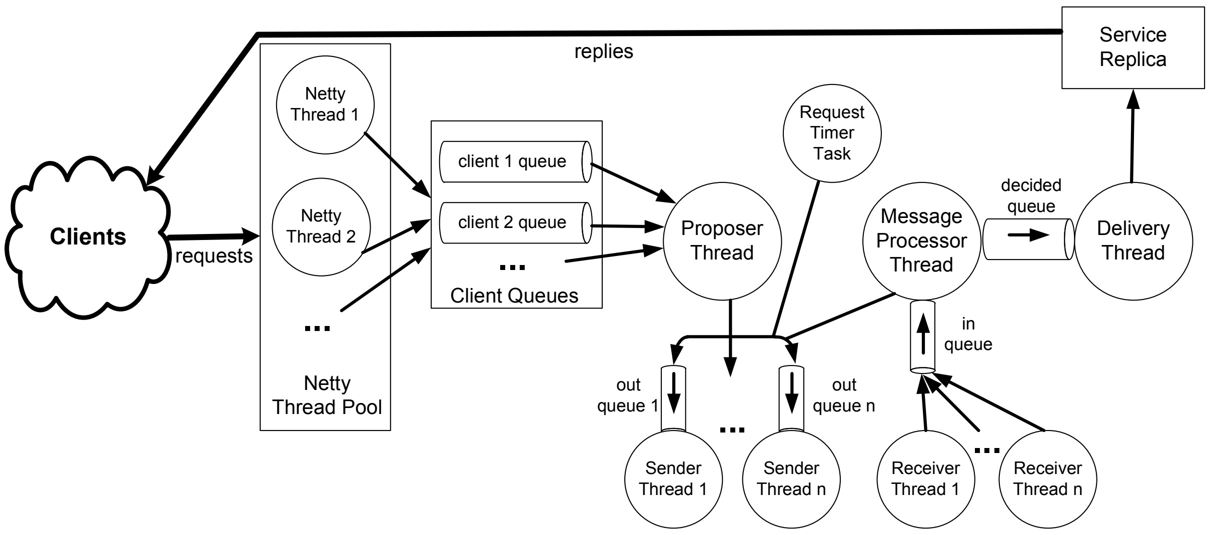


Figure 8.2: The staged message processing on BFT-SMaRt.

The client requests are received through a thread pool provided by the Netty communication framework. We have implemented a request processor that is instantiated by the framework and executed by different threads as the client load demands. The policy for thread allocation is at most one per client (to ensure FIFO communication between clients and replicas), and we can define the maximum number of threads allowed.

Once a client message is received and its MAC verified, we trigger the client manager that verifies the request signature and (if validated) adds it to the client's pending requests queue. Notice that since the signatures are verified by the *Netty threads*, multi-core and multi-processor machines would naturally exploit their power to achieve high throughput (verifying several client signatures in parallel).

The *proposer thread* will wait for three conditions before starting a new instance of the consensus: (i.) it is the leader of the first round of the next consensus; (ii.) the previous instance of the consensus is already finished; and (iii.) at least one client (pending requests) queue has messages to be ordered. In a leader replica, the first condition will always be true, and it will propose new requests to be ordered as soon as a previous consensus is decided and there are pending messages from clients. In non-leader replicas, this thread is always sleeping waiting for condition (i.).

Every message *m* to be sent by one replica to another replica is put on the *out queue* from which a *sender thread* will get *m*, serialize it, produce a MAC to be attached to the message and send it through TCP sockets. At the receiver replica, a *receiver thread* for this sender will read m, authenticate it (i.e., validate its MAC), deserialize it and put it on the *in queue*, where all messages received from other replicas are stored in order to be processed.

The *message processor thread* is responsible to process almost all messages of the state machine replication protocol. This thread gets one message to be processed and verifies if this message consensus is being executed or, in case there is no consensus currently being executed, it belongs to the next one to be started. Otherwise, either the message consensus was already finished and the message is discarded, or its consensus is yet to be executed (e.g., the replica is executing a late consensus) and the message is stored on the *out-of-context queue* to be processed when this future consensus is able to execute. If the message can be processed, its type will be evaluated (PROPOSE, WEAK, STRONG, etc.), and the quorum of received messages of this type is updated. If certain conditions are met (e.g., *there are more than* $\frac{n+f}{2}$ *WEAKs with the same value for this round*) new messages are produced and sent to other replicas (in the same way as explained before). As a side note, it is worth to mention that although the PROPOSE message contains the whole batch of messages to be ordered, the WEAKs and STRONGs messages only contain the hash of this batch.

When a consensus is finished on a replica (i.e., the replica received more than $\frac{n+f}{2}$ STRONGs for some value), the value of the decision is put on the *decided queue*. The *delivery thread* is responsible for getting decided values (a batch of requests proposed by the leader) from this queue, deserialize all messages from the batch, remove them from the corresponding client pending requests queues and mark this consensus as finalized. After that, the delivery thread invokes the service replica to make it execute and send replies to all request operations. The last thing done when processing this batch of requests is to store it on the state log or, if the checkpoint period was reached, to get the state from the service replica and clean the state log.

There are two other threads that sporadically can be activated to take actions on the protocol: the *request timer task* and *round timer task*. The latter is the timer associated to each round of the PaW protocol. When it is activated, it freezes the current round on the replica and tries to force a leader change to start a new round of the consensus instance [Zie04]. The request timer task by the other hand is activated periodically to verify if some request stayed more than a pre-defined timeout on the pending requests queue. The first time this timer expires for some request, causes this request to be forwarded to the current known leader. The second time this timer expires for some request, the instance currently running of the consensus protocol is stopped (if there is some running) and a RT-FREEZE message is sent to stop the consensus execution on all replicas and force a leader change.

Although we do not claim that the architecture depicted in Figure 8.2 is the best architecture for implementing state machine replication, it is the result of a three-year effort to make a high-throughput protocol in Java. During this effort, we tried several variants of this architecture, but in the end the one presented here represents the best approach we have found.

8.5 Using the BFT-SMaRt replication library

The basic use of the BFT-SMaRt replication library is quite simple. Figure 8.3 presents the client and replica classes that should be instantiated and extended, respectively, to implement a replicated service.

A BFT-SMaRt client just needs to instantiate the `ServiceProxy` class with a configuration file containing the IP, port, and public key of each replica. Then, whenever it needs to send a request to the replicas, it should call the `invoke` method and specify the request (serialized in a byte array) and indicate if it is a read-only request. At the server side, each replica must

```
//Client API
public class ServiceProxy ... {
  ...
  public byte[] invoke(byte[] command, boolean readOnly){
  ...
}

//Server API
public abstract class ServiceReplica ... {
  ...
  public abstract byte[] executeCommand(int clientId,
         long timestamp, byte[] nonces, byte[] command);
  public abstract byte[] getState();
  public abstract void setState(byte[] state);
}
```

Figure 8.3: API for using the BFT-SMaRt library at client and replica sides.

extend the `ServiceReplica` class and implement the abstract operations that will be called when there is a command to be executed or to get/set the state of the service. Notice that the `executeCommand` method provides also the id of the calling client, a timestamp, and a set of random nonces defined by the leader of the consensus that ordered this request. It is ensured that all replicas get equal timestamps and nonces associated to each request, which allows them to implement typical non-deterministic actions, such as reading the clock value and generating random numbers, in a deterministic way. Notice that read-only requests do not need to be ordered given that they should not change the service state (the Netty threads of Figure 8.2 deliver them directly to the service replica).

8.6 Performance

This section presents a preliminary performance evaluation of BFT-SMaRt version 0.4. Our objective here is not to present a full experimental analysis of the system (which is out of the scope of this practical experience report) but to present a basic assessment of the performance of it in the same framework as used by recent works on the area [CWA$^+$09a, VCBL09, CKL$^+$09].

Our setup is composed by a set of twelve 2.8 GHz Pentium-4 machines with 2 GBs RAM running Sun JDK 1.6 on top of Linux 2.6.18 connected by a Dell gigabit switch. In all experiments we enabled the Java Just-In-Time (JIT) compiler and run a warm-up phase to load and verify all classes, transforming the Java bytecodes into native code.

In the experiments we ran 4 and 7 machines as servers (for $f = 1$ and $f = 2$) and the remaining machines with 0 to 120 logical clients. We measured the *latency* and *throughput* of the system using a simple service with no state that executes null operations, using two request sizes: 4 and 1K bytes.

Table 8.1 shows the end-to-end average latency and peak throughput of BFT-SMaRt, PBFT[3] [CL02] and Spinning [VCBL09] considering $f = 1$ and small messages of 4 bytes.

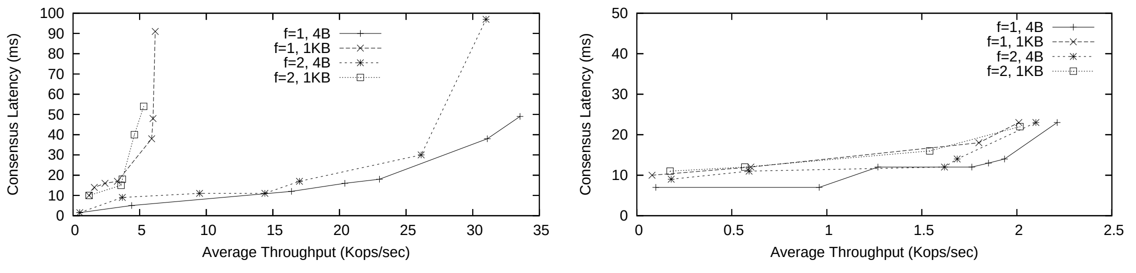
The table shows that although our system could not match the extremely low latency of the C-based PBFT it was able to present a throughput 45% better. More surprisingly, our throughput is better than the one obtained from the Spinning prototype, which was implemented in Java,

[3]Obtained from `http://www.pmg.csail.mit.edu/bft/`.

Metric	BFT-SMaRt	PBFT	Spinning
Latency	1.7 ms	0.4 ms	1.3 ms
Peak throughput	38 Kops/s	22 Kops/s	26 Kops/s

Table 8.1: Latency and peak throughput of SMaRt, PBFT and Spinning for requests with 4 bytes and $f = 1$.

runs several consensus in parallel, and uses a rotating leader, which distributes the load of proposing message batches between all replicas [VCBL09]. This ilustrates the effect that a well-designed implementation can have on the final values obtained from a protocol implementation.



(a) SHA1 MAC vectors (b) 1024-bit RSA signatures

Figure 8.4: Latency vs throughput of BFT-SMaRt with messages of 4 bytes and 1 Kbyte when $f = 1$ and $f = 2$.

Figure 8.4 shows BFT-SMaRt behaviour in different setups when under heavy load. Figure 8.4 (a) depicts the latency and throughput of BFT-SMaRt when client signatures are MAC vectors [CL02]. The figure reports average throughput values (measured in intervals of 100,000 requests) in which the protocol latency is under 100 ms, even when the system is highly loaded. These values were obtained using batches varying between 1 and 2K requests. The maximum average throughput reached was about 34 Kops/sec for small messages and 6 Kops/sec for larger messages.

The latency and throughput of BFT-SMaRt when client signatures are generated using 1024-bit RSA are shown in Figure 8.4 (b). This figure shows that our average throughput never surpasses 2.5 Kops/sec, which corresponds to the maximum number of signature verifications per second that our machines can do (a verification takes about 0.4 ms). In this case, the average throughput values were measured in intervals of 15,000 requests and we used batches varying between 1 and 100 requests.

Figure 8.4 shows also that moving from $f = 1$ to $f = 2$ does not affect substantially the average throughput achieved by the system with both types of signatures.

The machines used in the experiments described in this section are old and do not allow to explore the full potential of our highly modular architecture. We believe that by using current multi-core machines, the performance of BFT-SMaRt will improve, namely in terms of throughput. Notice however that, if the hardware is took into account, the current results are very good when compared with existing BFT replication libraries such as PBFT [CL02] or UpRight [CKL+09].

8.7 Lessons Learned

The three years of development of the two generations of BFT-SMaRt gave us important insights about how to implement high-performance fault-tolerant protocols in Java. In this section we discuss some of the lessons learned on this effort.

8.7.1 Making Java a BFT programming language

Despite the fact that the Java technology is used in most application servers and backend services deployed in enterprises, it is a common belief that a high-throughput implementation of a state machine replication protocol could not be possible in Java. We consider that the use of a type-safe language with several nice features (large utility API, no direct memory access, security manager, etc.) that makes the implementation of secure software more feasible is one of the key aspects to be observed when designing a replication library. For this reason, and because of its portability, we choose Java to implement BFT-SMaRt. However, our experience shows that these nice features of the language when not used carefully can cripple the performance of a protocol implementation. As an example, we will discuss how object serialization can be a problem.

One of the key optimizations that made our implementation efficient was to avoid Java default serialization in the critical path of the protocol. This was done in two ways: (1.) we defined the client-issued commands as byte arrays instead of generic objects, this removed the serialization and deserialization of this field of the client request from all message transmissions; and (2.) we avoid using object serialization on client requests, implementing serialization by hand (using data streams instead of object streams). This removed the serialization header from the messages and is specially important for client requests that are put in large quantities on batches to be decided by a consensus[4].

8.7.2 How to test BFT systems?

Testing BFT systems against malicious behaviours is tricky. The first challenge is to identify the critical malicious behaviours that should be injected on up to f replicas. The second challenge is how to inject the code of the malicious behaviours on these replicas. The first challenge can only be addressed with careful analysis of the protocol being implemented. Malicious code can be injected to the code using patches, aspect-oriented programming (through crosscutting concerns that can be activated on certain replicas) or simple commented code (which we are currently using).

It is worth to notice that most malicious behaviours can cause bugs that affect the liveness of the protocol, since basic mechanisms protect its safety (e.g., a leader proposing different values to different replicas should cause a leader change, not a disagreement). Moreover, the fact that the system tolerates arbitrary faults makes it mask some non-deterministic bugs, turning the whole test process even more difficult. For example, an older version of the BFT-SMaRt server communication system loosed some messages sporadically when under heavy load. The effect of this was that in certain conditions there was a leader change. We attributed that to asynchrony and request timeouts, however, after adopting a more disciplined test process, we found and corrected the bug.

[4]A serialized 0-byte operation client request requires 130 bytes with Java default serialization and 18 bytes in our serialization by hand.

8.7.3 Dealing with heavy loads

When testing BFT-SMaRt under heavy loads, we found several interesting behaviours that appear when a replication protocol is put under stress. The first one is that there are always f replicas that stay late in message processing. The reason is that only $n-f$ replicas are needed for the protocol to make progress and naturally f replicas will stay behind. A possible solution for this problem is to make the late replicas stay silent (and not load the faster replicas with late messages that will be discarded) and when they are needed (e.g., when one of the faster replicas fails) they synchronize themselves with the fast replicas using the state transfer protocol.

Another interesting point is that, in a switched network under heavy-load in which clients communicate with replicas using TCP, spontaneous total order (i.e., client requests reaching all replicas in the same order with high probability) almost never happens. This means that the synchronized communication pattern described in Figure 8.1 does not happen in practice. This same behaviour is expected to happen in wide-area networks. The main point here is that developers should not assume that client request queues on different replicas will be similar.

The third behaviour that commonly happens in several distributed systems is that their throughput tends to drop after some time under heavy-load. This behaviour is called *trashing* and can be avoided through a careful selection of the data structures[5] used on the protocol implementation and bounding the queues used for threads communication.

8.7.4 Signatures vs. MAC vectors

Castro and Liskov most important performance optimization to make BFT practical was the use of MAC vectors instead of public-key signatures. They solved a technological limitation of that time. In 2006, when we started developing BFT-SMaRt we avoided signatures at all costs due to the fact that the machines we had access at that time created and verified signatures much slowly than the machines we used in the experiments described in Section 8.6 (an RSA 1024-bit signature creation went from 15 ms to less than 5 ms while its verification went from 1 ms to less than 0.4 ms), and the high-end servers being packed today can do even better. We did some experiments in a 64-bit 2.3GHz quadcore Xeon machine and the signature verification takes less than 0.12 ms. In this same machine, the verification of a MAC takes about 0.01 ms. This means that with the machines available today, the problem of avoiding public-key signatures is not so important as it was a decade ago, specially if signature verification can be parallelized (as in our architecture). Moreover, there are two other already known advantages of using signatures [CWA+09a]: (1.) public key signatures on client requests makes it impossible for clients to forge MAC vectors and force leader changes; and (2.) if the service being replicated spends some non-negligible time answering a request, the throughput of a protocol that uses signatures becomes very similar to the throughput of some signature-free protocol since the crypto processing costs are diluted.

8.8 Concluding Remarks

This chapter reported our effort in building the BFT-SMaRt BFT state machine replication library. Our main objective here is to fill a gap in BFT literature describing how this kind of protocol can be implemented in a safe and performant way. The BFT-SMaRt system described

[5]For example, data structures that tend to grow with the number of requests being received should process searches in $\log n$ (e.g., using AVL trees) to avoid losing too much performance under heavy-load.

here is available as open source software in the project homepage [sma]. The road map found on this homepage describes a set of features and optimizations that we want to implement on the system. However, as our experiments show, the current implementation already provides a very good throughput for both small- and medium-size messages.

Chapter 9

Fault-tolerant workflow execution

Chapter Authors:
Johannes Behl (FAU), Klaus Stengel (FAU) and Rüdiger Kapitza (FAU)

9.1 Introduction

With the variety of cloud offerings increasing, ranging from very basic infrastructure services, such as storage, to more complex platform services (e.g., providing database-like functionality), to end-user directed solutions such as web-based office tools, there is a growing demand for a common way to manage their interaction. As most of these services can be accessed by standard web-service technology, business process management support such as provided by the Business Process Execution Language (BPEL) are an excellent match. Generic as well as domain-tailored offerings (e.g., Visual Process Manager[1] and runMyProcess[2]) are already available, and are becoming increasingly popular.

While most of them offer sophisticated interfaces, a multitude of connectors to subsystems, and support for non-functional properties such as scalability and security, are emerging, fault tolerance has so far received limited attention. However, recent studies on cloud offerings [SSR+10] and hardware in general [NDO11] show that clouds are less reliable than traditional data centers, and hardware faults are more common than previously assumed. In combination, this basically inhibits the outsourcing of critical processes (e.g., financial or medical services) to the cloud.

An effort to explicitly close this gap by supporting critical applications in the context of cloud computing is the EU IP project TClouds which targets the provision of a secure and resilient cloud infrastructure. As a part of this project, we present a flexible, lightweight, and cost-efficient approach to offer fault-tolerant execution of critical business processes at the platform-as-a-service (PaaS) level.

So far, common BPEL engines log their execution progress to stable storage in order to enable the recovery of long-running processes after a reboot or crash. However, this slows down the execution speed during operation and does not prevent the unavailability of services in case of a failure. Furthermore, fault tolerance must also be provided for the services called by a business process. This problem was initially addressed by Dobson, who analyzed how fault handling and replication can be integrated into a process description [Dob06]. Recently, Juhnke discussed how cloud resources can be exploited to tolerate infrastructure failures that affect the services of a process [JDF09]. Finally, Lau et al. [LLSFV08] proposed an initial approach

[1] http://www.salesforce.com/platform/process/
[2] http://www.runmyprocess.com/

to replicate a process engine by combining active and passive replication. This third solution requires a service engine to be executed at the client side, which complicates deployment and is costly in terms of resource demand and latency.

In contrast to these works, our approach provides fault tolerance by means of active replication at both the process level and at the service level. This is achieved by transforming a process definition dedicated for plain execution into a replication-aware version. This includes, that the handling of web-service calls is intercepted by custom proxy components that are co-located with the engine executing a replicated process instance. In this way, off-the-shelf engines can be used without modifications. To further simplify the task of replication, we use ZooKeeper [HKJR10b], a service to coordinate distributed applications. In addition to externalizing the coordination amongst the engine and service instances, we use ZooKeeper for configuration.

Our initial evaluation results show that our approach outperforms a BPEL engine that performs logging for recovery purpose by a factor of 2.0 to 3.4 while providing high availability. Furthermore, the overhead of externalizing the coordination is about 13% compared to a traditional design that utilizes a group communication as an integrated part co-located with the engines.

In the remainder of this paper, we give an overview on BPEL (Section 9.2). Section 9.3 outlines the proposed architecture and its realization. Section 9.4 details our evaluation results, and Section 9.5 concludes.

9.2 Basics About BPEL

The *Business Process Execution Language* (BPEL) is an XML-based language designed to describe and specify the behavior of *business processes*. In the context of BPEL, a business process is a set of activities that makes use of a composition of different web services (possibly from different providers) to provide a new, more complex web service.

Figure 9.1 shows a standard (unreplicated) BPEL infrastructure. Its main component is the *BPEL engine*, which is responsible for executing a business process specified in a *BPEL process definition*. Whenever a client sends a request to a BPEL service, the BPEL engine handles this request according to the corresponding process definition. In particular, it calls all web services necessary to fulfill the request.

Besides comprising the means to invoke web services, BPEL also provides mechanisms for holding intermediate data, handling exceptions, and expressing control flows which allow BPEL to be used to describe more complex business processes. Furthermore, most BPEL implementations rely on a recovery mechanism that logs intermediate state on a persistent storage to provide reliable process execution.

9.3 Reliable BPEL Infrastructure

Executing long-running critical tasks requires high availability. However, the standard BPEL mechanisms for fault tolerance support only the recovery from a machine crash but not continuous service provision despite faults. Additionally, BPEL processes depend on the web services they use, but fault tolerance of web services is not considered by standard BPEL infrastructures at all.

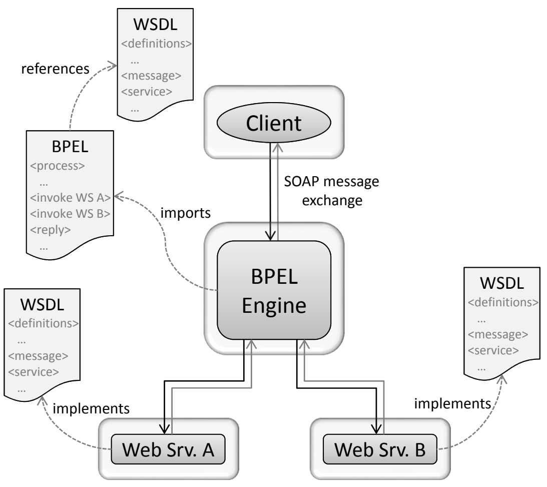


Figure 9.1: Standard unreplicated BPEL infrastructure.

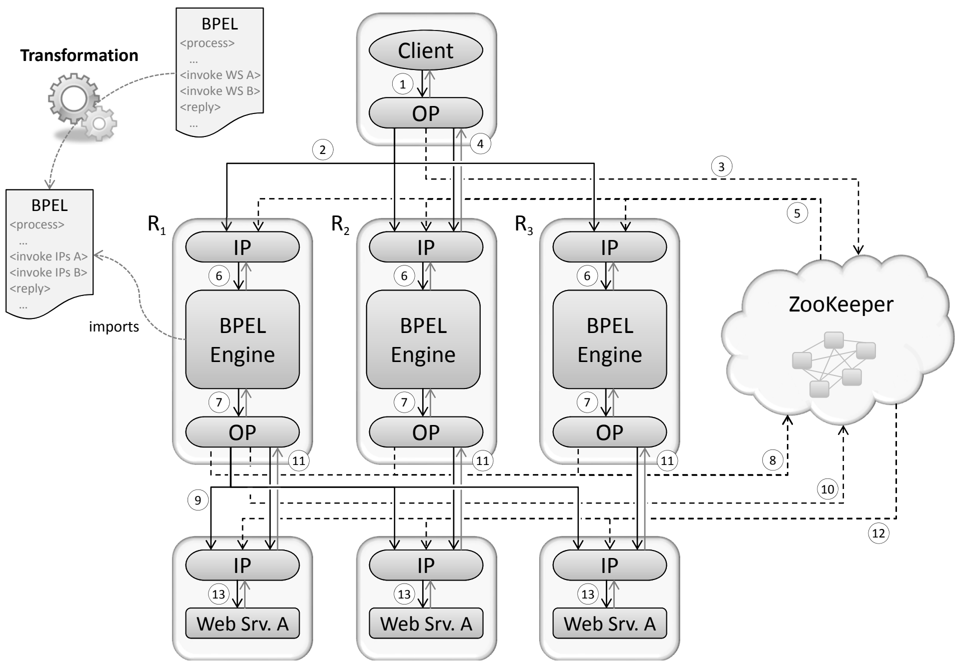


Figure 9.2: Proposed architecture: BPEL engine and web services are actively replicated using output and input proxies (OP and IP). An automatic transformation of process definitions allows them to intercept the communication chain transparently. The proxies make use of a ZooKeeper service for coordination and dynamic configuration.

9.3.1 Overall Architecture

We address this problem by actively replicating not only the BPEL engines but (optionally) also the web services in a combined architecture. This architecture (see Figure 9.2) is designed according to three main objectives: First, all measures taken for fault tolerance are to be transparent for the workflows described in BPEL; that is, it should not make a difference whether a process definition is executed on a standard or a replicated infrastructure. Second, all measures are to be as little invasive as possible for existing implementations in order to be able to reuse them. Third, for further minimizing the effort needed to provide a fault-tolerant solution, cloud services are to be used where possible.

In our architecture, these design objectives are met by means of proxies that intercept web-service calls to implement replication. In particular, the proxies distribute requests across replicas and collect results. Because web-service formats and protocols are used between clients and BPEL engines as well as between BPEL engines and web services, the replication of BPEL engines and web services can be achieved by almost the same mechanisms.

Moreover, the proxies use an external ZooKeeper service for coordination, dynamic retrieval of system information and configuration, crash detection and request ordering. Using an external service simplifies all these tasks, saves resources and permits a global coordination within clouds infrastructures.

Transparent interception of web-service calls at the client can be implemented by inserting the proxy into the web-service library. However, this is not possible on the side of the replicated BPEL engine, at least if existing BPEL systems should not be modified. Therefore, we establish a transformation of process definitions, which reroutes the web-service calls. This transformation is done automatically before the process definition is loaded by a BPEL engine.

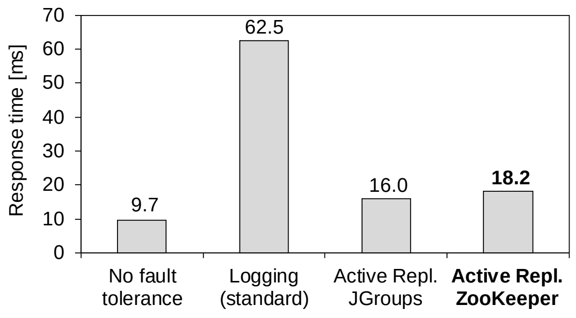
9.3.2 Implementation Details

In the following, we present the steps necessary to process a client request in our architecture.

Client/BPEL Stage When a client sends a request using a web-service library, the request is passed to a local *output proxy* (①, see Figure 9.2). The proxy is responsible for assigning unique ids to the requests, sending the request data to all BPEL engine replicas ②, registering the request at ZooKeeper ③ and for collecting the results ④. ZooKeeper is used at this stage to obtain the active replicas and to enforce a total order on all requests. The former allows an easy propagation of configuration changes within the system, for example, if replicas fail or are replaced. The latter guarantees that all BPEL engine replicas process the same sequence of input data. Sending the request data in a separate step prior to the registration is done for performance reasons, as ZooKeeper is designed for use with small chunks of data. After distributing and registering the request, the output proxy waits until a reply becomes available.

In our architecture, output proxies do not communicate directly with BPEL engines. Instead, each replica runs an *input proxy* in front its BPEL engine. Input proxies maintain a steady connection to ZooKeeper, which allows the detection of crashed replicas. Moreover, they are informed by ZooKeeper when a new request has been registered ⑤. In that case, they deliver the corresponding request data to the local BPEL engine ⑥ and wait for the reply. After the engine has processed the request, an input proxy stores the reply in a local cache where it can be retrieved by the output proxy of the client.

(a) BPEL echo process called by a client (Client/BPEL stage)



(b) Echo web service called by BPEL process (BPEL/WS st.)

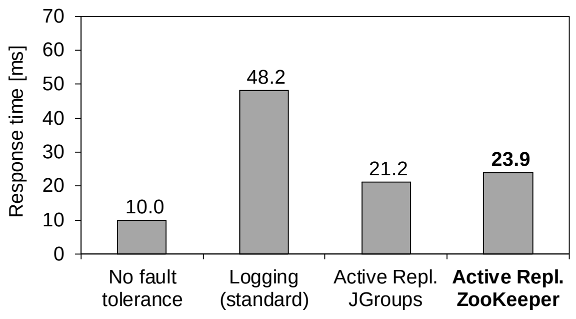


Table 9.1: Architecture comparison: Unreplicated with and without logging-based fault tolerance vs. transparent active replication via ZooKeeper and JGroups (times are in ms).

BPEL/Web-Service Stage As on the client side, each outgoing call of a BPEL engine replica is intercepted by an output proxy. Here, the transformation of process definitions (see Section 9.3.1) ensures that outgoing calls are rerouted suitably ⑦. Furthermore, the transformation process tags each call with an id. In doing so, special situations, such as loops and the parallel execution of the same workflow, have to be considered. Uniquely identifying the invocation of a web service is mandatory, since that enables output proxies to distinguish different calls and to prevent multiple executions of a single web-service call.

After the output proxies have been called by the local BPEL engines, they determine by means of ZooKeeper and the tagged call id, whether the specific invocation is already being conducted by another output proxy ⑧. If this is not the case, the corresponding output proxy performs the actual call. Otherwise, the output proxy randomly chooses a web-service replica for retrieving the cached reply. Thus, steps ⑨ to ⑬ are similar to steps ② to ⑥.

Relying on ZooKeeper allows the output proxies of BPEL engine replicas to detect if the replica that performs the actual web-service call has crashed. In this case, the remaining BPEL engine replicas use ZooKeeper's support for leader election to select a replica to complete the invocation.

9.4 Evaluation

In the course of the evaluation, we pursue two basic questions: How does the proposed transparent active replication perform compared to a standard BPEL infrastructure that uses logging, and what are the costs of externalizing coordination among replicas. To answer theses questions, we compare our approach with a standard unreplicated BPEL engine and with a version featuring a traditional fault-tolerant architecture using a group communication framework.

Test Cases In the case of the traditional fault-tolerant architecture, output proxies send request data to only one of the input proxies and not to all of them (cf. steps ② and ⑨). The reliable delivery and ordering of request data is subsequently carried out using a group communication framework, in our case JGroups. Additional steps (see ③ and ⑩) as needed in the ZooKeeper variant can be omitted.

As a single request from a client to a BPEL engine typically leads to interaction with multiple web services, we examine these two stages independently. In the first scenario, we implement an echo service purely in BPEL. After the corresponding BPEL process has been started, a client sends a message, which is then immediately answered by the process. In the second scenario, we implement a similar echo service as a web service. Here, we use a BPEL process for measuring the elapsed time while calling this web service from the BPEL engine.

Hard- and Software Setting All measurement results are obtained on the basis of a test installation comprising 16 hosts equipped with 2.4 GHz quad-core CPU, 8 GB RAM, and connected over Gigabit switched Ethernet. As platform for BPEL engines and web services, we use an Apache software stack containing Tomcat, Axis2 and ODE. The replication groups of BPEL engines, web services, and ZooKeeper comprise five servers[3] each and are therefore able to tolerate two faults per replica group. The values presented are the average of several test runs; each test run includes 250 requests.

Results Figure 9.1 illustrates the performance advantages of a replication-based solution compared to the standard approach using logging, which enables a BPEL process to survive a reboot due to a crash or maintenance. Using ZooKeeper-aided active replication to provide a fault-tolerant BPEL implementation, response times in our approach are 3.4 times (Client/BPEL stage) and 2.0 times (BPEL/web services stage) lower compared to a standard BPEL infrastructure. Here, the slightly less performance gain at the stage between BPEL process and web service is mainly owed to the support of transparent replication in our architecture and to the additional web-service call it leads to (see Figure 9.2, step ⑦).

Besides achieving higher performance than a standard unreplicated BPEL implementation, our fault-tolerant BPEL infrastructure presented here also provides improved fault tolerance: The services offered by a replicated business process remain available even in the presence of a limited number of crashes.

Disabling the logging mechanism for the unreplicated BPEL engine (i.e., deactivating fault tolerance entirely) exposes the high costs of this technique. Without logging state to persistent storage, response times of the unreplicated BPEL engine drop to about 10 milliseconds for both stages. However, improving performance this way is not acceptable for critical services, as it prevents a service from being recovered after a crash.

Considering the second question guiding this evaluation, it can be stated that the overhead of using external coordination is moderate (about 13%). Despite the increased number of exchanged messages, the extra costs in terms of latency are at least partially hidden due to concurrent execution. Even if the underlying network exhibited higher latencies the overhead should

[3]Five machines are a common configuration for ZooKeeper.

be negligible in most cases. In sum, this makes using coordination as an external service attractive, as it is often already available in cloud infrastructures and data centers, thereby saving on deployment and maintenance costs.

9.5 Conclusion

We presented a flexible, lightweight and cost-efficient approach to support the fault-tolerant execution of critical business processes in a cloud setting. This is achieved by off-line transformation of processes, thereby making them replication-aware and the rigorous use of ZooKeeper to handle coordination and configuration. Our approach incurs only moderate overhead compared with a traditional fault-tolerant architecture in which a group communication facility is directly integrated with the middleware and it outperforms an unreplicated BPEL execution supporting only the recovery from crashes.

Chapter 10

BFT MapReduce

Chapter Authors:
Alysson Bessani (FFCUL), Marcelo Pasin (FFCUL), Miguel Correia (FFCUL) and Pedro Costa (FFCUL)

10.1 Introduction

MapReduce is a programming model and a runtime environment designed by Google for processing large data sets in its *warehouse-scale machines* (WSM) with hundreds to thousands of servers [DG04, HB09]. MapReduce is becoming increasingly popular with the appearance of many WSMs to provide *cloud computing* services, and many applications based on this model. This popularity is also shown by the appearance of open-source implementations of the model, like Hadoop that appeared in the Apache project and is now extensively used by Yahoo and many other companies [Whi09].

At scales of thousands of computers and hundreds of other devices like network switches, routers and power units, component failures become frequent, so fault tolerance is central in the design of the original MapReduce as also in Hadoop. The modes of failure tolerated are reasonably benign, like component crashes, and communication or file corruption. Although the availability of services based on these mechanisms is high, there is anecdotal evidence that more pernicious faults do happen and that they can cause service unavailabilities. Examples are the Google App Engine outage of June 17, 2008 and the Amazon S3 availability event of July 20, 2008.

This combination of the increasing popularity of MapReduce applications with the possibility of fault modes not tolerated by current mechanisms suggests the need to use fault tolerance mechanisms that cover a wider range of faults. A natural choice is Byzantine fault-tolerant replication, which is a current hot topic of research but that has already been shown to be efficient [KAD$^+$07, VCBL09]. Furthermore, there are critical applications that are being implemented using MapReduce, as financial forecasting or power system dynamics analysis. The results produced by these applications are used to take critical decisions, so it may be important to increase the certainty that they produce correct outputs. Byzantine fault-tolerant replication would allow MapReduce to produce correct outputs even if some of the nodes were arbitrarily corrupted. The main challenge is doing it at an affordable cost, as BFT replication typically requires more than triplicating the execution of the computation [KAD$^+$07].

10.2 Hadoop MapReduce

MapReduce is used for processing large data sets by parallelizing the processing in a large number of computers. Data is broken in splits that are processed in different machines. Processing is done in two phases: *map* and *reduce*. A MapReduce application is implemented in terms of two functions that correspond to these two phases. A map function processes input data expressed in terms of key-value pairs and produces an output also in the form of key-value pairs. A reduce function picks the outputs of the map functions and produces outputs. Both the initial input and the final output of a Hadoop MapReduce application are normally stored in HDFS [Whi09], which is similar to the Google File System [GGL03]. Dean and Ghemawat show that many applications can be implemented in a natural way using this programming model [DG04].

A MapReduce *job* is a unit of work that a user wants to be executed. It consists of the input data, a map function, a reduce function, and configuration information. Hadoop breaks the input data in *splits*. Each split is processed by a map task, which Hadoop prefers to run in one of the machines where the split is stored (HDFS replicates the splits automatically for fault tolerance). Map tasks write their output to local disk, which is not fault-tolerant. However, if the output is lost, as when the machine crashes, the map task is simply executed again in another computer. The outputs of all map tasks are then merged and sorted, an operation called *shuffle*. After getting inputs from the shuffle, the reduce tasks process them and produce the output of the job.

The four basic components of Hadoop are: the *client*, which submits the MapReduce job; the *job tracker*, which coordinates the execution of jobs; the *task trackers*, which control the execution of map and reduce tasks in the machines that do the processing; HDFS, which stores files.

10.3 BFT Hadoop MapReduce

We assume that clients are always correct. The rationale is that if the client is faulty there is no point in worrying about the correctness of the job's output. Currently we also assume that the job tracker is never faulty, which is the same assumption done by Hadoop [Whi09]. However, we are considering removing this restriction in the future by replicating also the job tracker using BFT replication. In relation to HDFS, we do not discuss here the problems due to faults that may happen in some of its components. We assume that there is a BFT HDFS, which in fact has already been presented elsewhere [CWA+09b]. Task trackers are present in all computers that process data, so there are hundreds or thousands of them and we assume that they can be Byzantine, which means that they can fail in a non-fail-silent way.

The key idea of BFT Hadoop's task processing algorithm is to do majority voting for each map and reduce task. Considering that f is a high bound on the number of faulty task trackers, the basic scheme is the following:

1. start $2f+1$ replicas of each map task; write the output of these tasks to HDFS;
2. start $2f+1$ replicas of each reduce task; processing in a reduce starts when it reads $f+1$ copies of the same data produced by different map replicas for each of map task; the output of these tasks is written to HDFS.

This basic scheme is straightforward but is also inefficient because it multiplies the processing done by the system. Therefore, we use a set of improvements:

Reduction to $f+1$ replicas. The job tracker starts only $f+1$ replicas of the same task and the reduce tasks check if all of them return the same result. If a timeout elapses or some of the returned results do not match, more replicas (at most f) are started, until there are $f+1$ matching replies.
Tentative execution. Waiting for $f+1$ matching map results before starting a reduce task can put a burden on end-to-end latency for the job completion. A better way to deal with the problem is to start executing the reduce tasks just after receiving the first copies of the required map outputs, and then, while the reduce is still running, validate the input used as the map replicas outputs are produced. If at some point it is detected that the input used is not correct, the reduce task can be restarted with the correct input.
Digest replies. We need to receive at least $f+1$ matching outputs of maps or reduces to consider them correct. These outputs tend to be large, so it is useful to fetch the first output from some task replica and get just a digest (hash) from the others. This way, it is still possible to validate the output without generating much additional network traffic.
Reducing storage overhead. We can write the output of both map and reduce tasks to HDFS with a replication factor of 1, instead of 3 (the default value). We are already replicating the tasks, and their outputs will be written on different locations, so we do not need to replicate these outputs even more. In the normal case Byzantine faults do not occur, so these mechanisms greatly reduce the overhead introduced by the basic scheme. Specifically, without Byzantine faults, only $f+1$ replicas are executed in task trackers, the latency is similar to the one without replication, the overhead in terms of communication is small, and the storage overhead is minimal.

10.4 Conclusion and Future Work

This abstract briefly presents a solution to make Hadoop MapReduce tolerant to Byzantine faults. Although most BFT algorithms in the literature require $3f+1$ replicas of the processing, our solution needs only $f+1$ in the normal case, in which there are no Byzantine faults.

Currently we are implementing a prototype of the system, which we will evaluate it in a realistic system to see if the actual costs match our expectations.

Chapter 11

Logging

Chapter Authors:
Davide Vernizzi (POL), Emanuele Cesena (POL), Gianluca Ramunno (POL) and Paolo Smiraglia (POL)

11.1 Introduction

In our vision of cloud, the presence of a efficient logging system is necessary. In this context we define the Log Service, a service which has as objective to track and to log events that happen in the cloud at different levels. In this chapter we analyse how the Log Service can be put in a multi-cloud context.

The structure of the chapter includes two sections. The former provides a background about Log Service and the latter provides a discussion about three applications of Log Service in a multi-cloud context.

11.2 Log Service for a single cloud

The main focus of Log Service is to log and track events originating at the infrastructure layer. This service is mainly based on the scheme for secure logging proposed by Schneier and Kelsey in [SK99]. To ensure the code-integrity and hence to enhance the trustworthiness, in Log Service the Trusted Computing technology is used. For a more detailed description of Log Service, see D2.1.1, Chapter 6.

Log entries are usually small pieces of data, created at high rate and rarely deleted. For these reasons, the Log Service must be capable of recording many log entries and of providing a view on a subset of the log entries that satisfy a particular query. Moreover, in order to ensure the security of the log entries, the Log Service must be capable to guarantee their integrity and confidentiality. Since the log entries may contain sensitive information about the usage of a certain system, Log Service must be capable to mediate every access in order to prevent leakage of information. Therefore the presence of an access control system is required.

Log Service may be considered as the ensemble of three components. The *Log Core* which is the main component and acts as service controller, the *Log Storage* which manages the storage of the log entries and the *Log Console* which acts as public access interface to the Log Service. Moreover, in Log Service four actors may be identified. *Cloud Component* is a generic component of the cloud infrastructure, *User* is the end-user of the cloud services, *Cloud Admin* is the administrator of the cloud infrastructure and *Log Reviewer* is an external entity which is

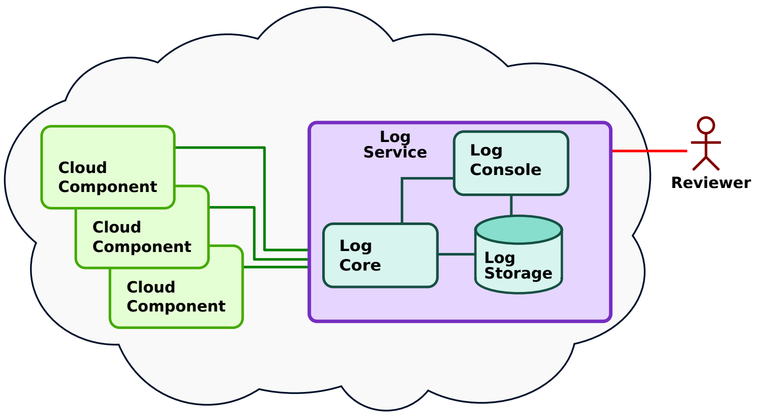


Figure 11.1: High level architecture of the Log Service.

able to read the logs. The relationships between components and actors are depicted in figure 11.1.

11.3 Multi-cloud scenarios for the Log Service

For the *Log Service* we foresee different multi-cloud scenarios that represent possible steps along evolutionary paths from a service entirely confined within a single cloud (described in D2.1.1 and briefly recalled in Section 11.2) – the *TClouds* cloud – to a completely distributed one. In particular, we see two different and, possibly, orthogonal directions for enhancements: moving the clients of the Log Service to remote clouds and logging Cloud-of-Cloud events (i.e. originated by BFT protocols). In the first direction we identify one generic scenario called *Log as a Service* (Section 11.3.1). In the other direction we identify three *Log Service* scenarios: *Cloud-of-Cloud-enabled* (Section 11.3.2), *Cloud-of-Cloud-optimized* (Section 11.3.3) and *Cloud-of-Cloud-distributed* (Section 11.3.4).

11.3.1 Log as a Service (LaaS)

The starting point for the *Log as a Service* (LaaS) scenario is a *Log Service* (LS) designed for single cloud, a trustworthy service whose clients are internal (i.e. in the same cloud). They are (physical and virtual) infrastructural components and applications, therefore the LS can be considered as part of the *Infrastructure as a Service* (IaaS) and also of the *Platform as a Service* (PaaS). The LS can be extended to the Log as a Service by moving the location of the clients originating the log events: in LaaS they are external to the cloud running the LS, i.e. they are from remote clouds. However, with respect to the client types, the LaaS keeps serving both infrastructure and application clients.

A first example of client in this scenario could be a remote *private* cloud that wants to outsource the management and the storage of the log events originated at the infrastructure layer to an external cloud entirely dedicated to this purpose (Figure 11.2). Such a service (LaaS) is already available on the market (e.g. Loggly[1]) which is particularly focused on the analysis of logs collected from multiple sources. The aim of TClouds LaaS is to provide similar functional capabilities – although the focus will not be the log analysis and the related tools –

[1]Loggly - http://loggly.com

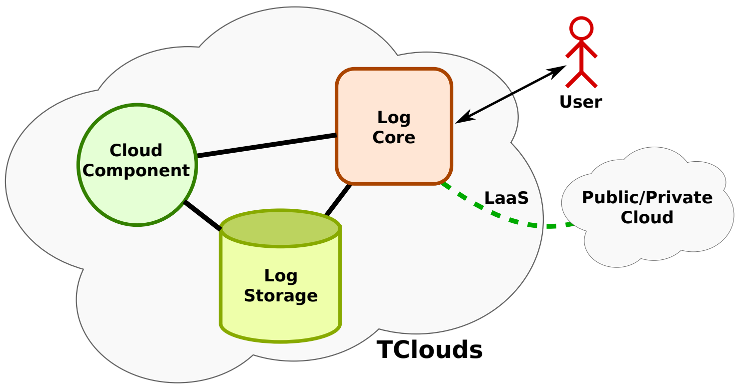


Figure 11.2: *Log as a Service* scenario.

and, at the same time, to guarantee that a number of security requirements like forward integrity, confidentiality, access control and privacy by design.

In this example the remote cloud is supposed to be mainly private because a commercial provider would not expose the internals of the cloud it operates by outsourcing the storage of log events or allowing a customer to do so. Moreover, given the high frequency of the infrastructural log events and the large size of the infrastructure of a public cloud, a massive amount of log data would be transferred to the external Log Service cloud, requiring large bandwidth only dedicated to this purpose. This would also imply relevant economic costs, if the outsourcing can be decided by the customer, due to the communication cost models common to the commercial providers. However also a *public* cloud might be a customer of the LaaS. For instance when the latter acts as *Platform as a Service* to log application events. In this case the customer of the public cloud can autonomously decide to outsource the collection of the log events originated from its application(s).

Specific requirements

In the following some important aspects of the Log as a Service are presented by highlighting, for each of them, the differences from the Log Service designed in D2.1.1 for a single cloud.

Functional model. The Log as a Service builds on a functional model similar to the one underlying the Log Service. Only one role differs: the Cloud Nodes originating the log events – both infrastructural and applicative – may be not running in the same cloud as the Log as a Service: these physical nodes can be local or remote.

Trust model. The Log as a Service builds on the same trust model as the Log Service. The remote cloud accesses the service exported by a cloud which is considered trustworthy.

Authentication. The authentication mechanism provided by the Log Service should be extended to support the identification of the cloud originating the log events in addition to the originator node and the user account. For instance, inter-domain authentication mechanisms like *OAuth*[2] may be applied.

[2]OAuth - http://oauth.net/

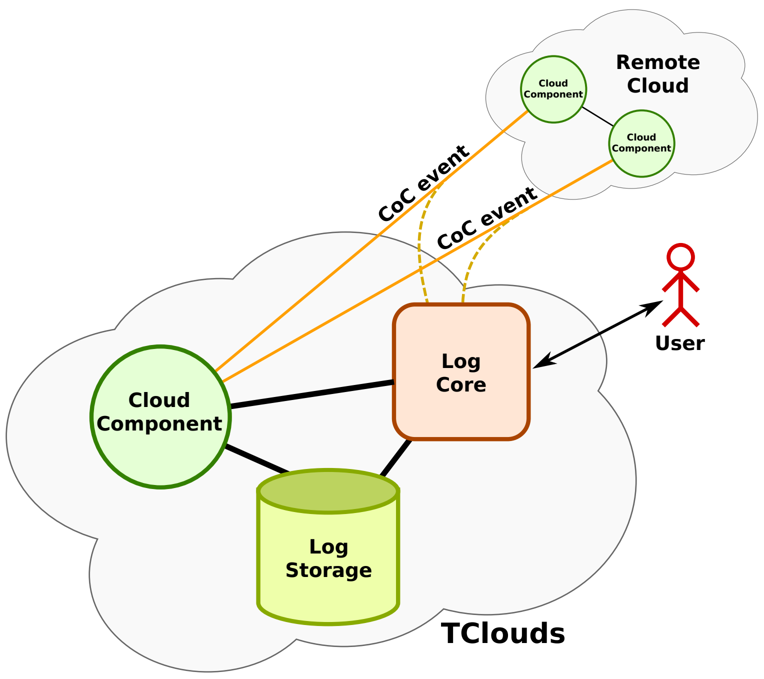


Figure 11.3: *Cloud-of-Cloud-enabled Log Service* scenario.

Access control. Analogously, the access control mechanism embedded in the Log Service should be extended to encode the identification of the cloud originating the log events in addition to the originator node and the user account.

Storage. For the storage, there is no difference between Log Service and Log as a Service: the storage of the log events is completely confined within a single cloud, the one where either log service types runs.

11.3.2 Cloud-of-Cloud-enabled Log Service

The *Cloud-of-Cloud-enabled Log Service* is the first example of a multi-cloud application of Log Service. In this context the objective is to track and log Cloud-of-Cloud events, for instance events generated by BFT protocols (see, D2.2.1 Section 3.3). In this scenario we consider a Cloud-of-Cloud which is composed by one trustworthy cloud (TCloud) and less trusted clouds (Figure 11.3). In addiction, we also assume that in this architecture, a fault tolerance system (e.g. built upon BFT protocols) is enabled. Considering the scenario, an analysis of the most important aspects of Cloud-of-Cloud-enabled Log Service is now presented.

Access Control. In a Cloud-of-Cloud architecture, each cloud may be considered as an access domain which is characterized by a certain access level to the information. Moreover, we consider that such a level may vary depending on the location of the cloud (e.g., logs about medical data must only be accessed in the same country where the corresponding data are generated). Therefore, in order to ensure the privacy, Cloud-of-Cloud-enabled Log Service must be able to regulate the access to the stored data and to manage correctly the variation of the access level.

Event-driven logging. In a Cloud-of-Cloud where BFT protocols are enabled, each event implies the exchange of a large amount of messages between nodes which may be part of

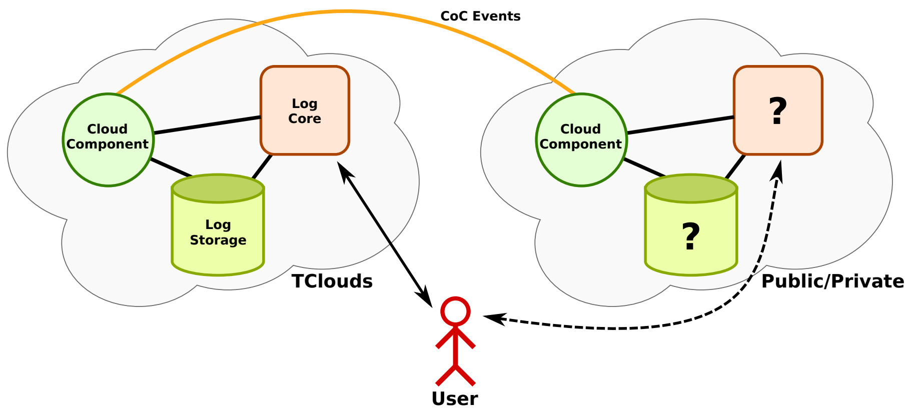


Figure 11.4: *Cloud-of-Cloud-optimized Log Service* scenario.

different clouds. These messages may be used as trigger for the identification of a certain event. This way, by defining a proper communication protocol among Cloud-of-Cloud nodes, it is possible to realize an event-driven logging system.

Forward Integrity. A key point for Log Service is to ensure the Forward Integrity property. While in a single-cloud scenario the mechanisms to ensure this property are well defined (see D2.1.1, Section 6.3), in a multi-cloud scenario some issues may arise. The first one is how to guarantee the Forward Integrity between multiple log files which come from different sources. In order to mitigate this issue a proper schema based on the usage of hash-chains must be defined. These schemes usually makes use of cryptographic keys to provide confidentiality and integrity. The generation and the distribution of such keys may bring security issues, especially in distributed environments. To address this, specific techniques (such as key-splitting) will be applied.

11.3.3 Cloud-of-Cloud-optimized Log Service

Cloud-of-Cloud-optimized Log Service may be considered the first optimization step of the Cloud-of-Cloud-enabled Log Service scenario. In this context the Log Service not only runs on the trustworthy cloud, but it can be also distributed on different clouds. The "distributed" aspect of Log Service means that some components like Log Core (or Log Storage) can be replicated on a different cloud nodes (which may be part of untrusted clouds) in order to build, for instance, a fault tolerance system (see Figure 11.4).

Cooperation between multiple components. The first aspect that requires a discussion is the cooperation between multiple replicas of the components. Replicas of the log components may be used to realize a fail-over system or to create a remote architecture of a system with a local replica of the components. The cooperation between multiple components brings to an issue related to the location where the component replicas may be positioned. In detail, some security risks may appear if the replicas are located in "hostile" environments, for instance if a replica of Log Core is running in a portion of a public cloud.

Management Optimization. Another aspect of the Cloud-of-Cloud-optimized Log Service is the optimization of the replicas management. In detail, in order to enhance the cloud performance it may be useful to identify algorithms and policy to determine *when* the usage of the

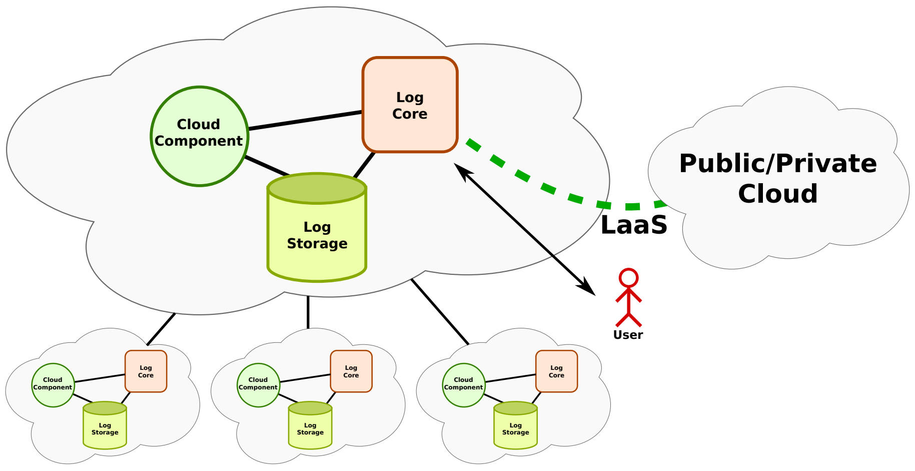


Figure 11.5: *Cloud-of-Cloud-distributed Log Service* scenario.

replicas is necessary and *where* they have to be located into the Cloud-of-Cloud. Concerning the location of the replicas, some issues may be related to the legislation of the region where the cloud is located (see D1.2.2 for more details).

11.3.4 Cloud-of-Cloud-distributed Log Service

Cloud-of-Cloud-distributed Log Service can be viewed as a fully multi-cloud version of LaaS. In this scenario the Log Service is running on a Cloud-of-Cloud architecture and it is exported as service (see Figure 11.5).

In this case all the components of the log service take benefits from the underlying distributed infrastructure. Here, differently from the solution presented in the previous sections, the LaaS is seen as a service which is self-managed and self-coordinated and which completely relies on the underlying highly distributed infrastructure (e.g. it does not require the presence of a trusted cloud to run the Log Core). Since this scenario does not rely a trustworthy cloud, it is necessary to guarantee security of cryptographic keys and sensitive data.

11.4 Concluding Remarks

This chapter described how the Log Service presented in D2.1 can take advantage of the architecture of the cloud of clouds. The Log service is enhanced along three main lines.

First, it is offered as Log as a Service (LaaS) to physical nodes or applications running in other clouds which want to outsource the management and the storage of the logs. Second, the LaaS can be used to track events generated by the BFT protocols that are executed in the cloud of clouds. Finaly, the LaaS can exploit the distributed nature of the cloud of clouds in order to achieve higher availability and confidentiality.

In all the cases, special attention to the security issues must be paid. When accessing the LaaS from many different domains, it is important to guarantee that a proper access control mechanism is in place. On the other hand, when the LaaS exploits remote clouds (e.g., for creating a BFT service), it is necessary to protect the cryptographic keys used to guarantee integrity and confidentiality of the log entries.

Bibliography

[ABO03] Hagit Attiya and Amir Bar-Or. Sharing memory with semi-Byzantine clients and faulty storage servers. In *Proc. of the 22rd IEEE Symposium on Reliable Distributed Systems - SRDS 2003*, pages 174–183, October 2003.

[AC77] Algirdas Avizienis and L. Chen. On the implementation of N-version programming for software fault tolerance during execution. In *Proceedings of the IEEE COMPSAC*, pages 149–155, Chicago, IL, USA, November 1977.

[ACKL08] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Byzantine replication under attack. In *Proc. of the Int. Conf. on Dependable Systems and Networks – DSN'08*, jun 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, April 2006.

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

[ALPW10] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. RACS: A case for cloud storage diversity. *Proc. of the 1st ACM Symposium on Cloud Computing*, pages 229–240, June 2010.

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

[BACF08] Alysson N. Bessani, Eduardo P. Alchieri, Miguel Correia, and Joni S. Fraga. DepSpace: a Byzantine fault-tolerant coordination service. In *Proc. of the 3rd ACM European Systems Conference – EuroSys'08*, pages 163–176, April 2008.

[BEG$^+$94] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. *Algorithmica*, 12:225–244, 1994.

[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.

[BMST93] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. *Distributed Systems*, chapter The Primary-Backup Approach, pages 199–216. ACM Press, 1993.

[Bru10] Grant Brugher. Secure use of cloud storage. Blackhat Briefings USA 2010, July 2010. Version 1.0 (final).

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

[CDE+10] Serdar Cabuk, Chris I. Dalton, Konrad Eriksson, Dirk Kuhlmann, HariGovind V. Ramasamy, Gianluca Ramunno, Ahmad-Reza Sadeghi, Matthias Schunter, and Christian Stuble. Towards automated security policy enforcement in multi-tenant virtual data centers. *Journal of Computer Security*, 18(1):89–121, January 2010.

[CG09] Christian Cachin and Martin Geisler. Integrity protection for revision control. In Michel Abdalla and David Pointcheval, editors, *Proc. Applied Cryptography and Network Security (ACNS)*, volume 5536 of *Lecture Notes in Computer Science*, pages 382–399. Springer, 2009.

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

[CKL+09] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor RichÈ. UpRight cluster services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles – SOSP'09*, October 2009.

[CKS09] Christian Cachin, Idit Keidar, and Alexander Shraer. Fail-aware untrusted storage. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 494–503, 2009.

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

[CLNV02] Miguel Correia, Lau Cheuk Lung, Nuno F. Neves, and Paulo Veríssimo. Efficient Byzantine-resilient reliable multicast on a hybrid failure model. In *Proceedings of the 21st Symposium on Reliable Distributed Systems*, Suita, Japan, October 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.

[CML+06] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ-Replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI'06*, November 2006.

[CMSK07] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proc. of the 21st ACM Symposium on Operating Systems Principles – SOSP'07*, October 2007.

[CNV04] Miguel Correia, Nuno F. Neves, and Paulo Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proc. of the 23rd IEEE Symposium on Reliable Distributed Systems - SRDS 2004*, October 2004.

[CRL03] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions Computer Systems*, 21(3):236–269, August 2003.

[csa09] Cloud security alliance — security guidance for critical areas of focus in cloud computing (v2.1). http://www.cloudsecurityalliance.org/, 2009.

[CSS07] Christian Cachin, Abhi Shelat, and Alexander Shraer. Efficient fork-linearizable access to untrusted shared memory. In *Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 129–138, 2007.

[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+09a] 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.

[CWA+09b] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. UpRight cluster services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, October 2009.

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

[DK11] Tobias Distler and Ruediger Kapitza. Increasing performance in Byzantine fault-tolerant systems with on-demand replica consistency. In *Proceedings of the 6th ACM SIGOPS/EuroSys European Systems Conference - EuroSys'11*, April 2011.

[DLS88] Cyntia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–322, April 1988.

[Dob06] Glen Dobson. Using WS-BPEL to implement software fault tolerance for web services. In *Proc. of the 32nd Euromicro Conf. on Software Engineering and Advanced Applications*, pages 126–133, 2006.

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

[ES05] Richard Ekwall and André Schiper. Replication: Understanding the advantage of atomic broadcast over quorum systems. *Journal of Universal Computer Science*, 11(5):703–711, 2005.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FZFF10a] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation – OSDI'10*, pages 337–350, October 2010.

[FZFF10b] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, 2010.

[GBG+11] Miguel Garcia, Alysson Bessani, Illir Gashi, Nuno Neves, and Rafael Obelheiro. OS diversity for intrusion tolerance: Myth or reality? In *Proc. of the Int. Conf. on Dependable Systems and Networks – DSN'11*, Hong Kong, 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 Gifford. Weighted voting for replicated data. In *Proc. of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, December 1979.

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

[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.

[GPS07] Ilir Gashi, Peter Popov, and Lorenzo Strigini. Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers. *IEEE Transactions on Dependable and Secure Computing*, 4(4):280–294, Oct./Dec. 2007.

[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*, 2010.

[Gro07a] Trusted Computing Group. Tpm main, part 1 design principles. specification version 1.2, revision 103. July 2007.

[Gro07b] Trusted Computing Group. Tpm main, part 3 commands. specification version 1.2, revision 103. July 2007.

[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.

[HB09] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[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 Trans. on Programing Languages and Systems*, 13(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.

[HKJR10a] Patrick Hunt, Mahadev Konar, Flavio Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for Internet-scale services. In *Proc. of the USENIX Annual Technical Conference – ATC 2010*, pages 145–158, June 2010.

[HKJR10b] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proc. of the 2010 USENIX Annual Technical Conf.*, pages 145–158, 2010.

[HLM03] Maurice Herlihy, Victor Lucangco, and Mark Moir. Obstruction-free syncronization: double-ended queues as an example. In *Proc. of the 23th IEEE Int. Conference on Distributed Computing Systems - ICDCS 2003*, pages 522–529, July 2003.

[HW90a] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems*, 12(3):463–492, July 1990.

[HW90b] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[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.

[JDF09] Ernst Juhnke, Tim Dörnemann, and Bernd Freisleben. Fault-tolerant BPEL workflow execution via cloud-aware recovery policies. In *Proc. of the 35th Euromicro Conf. on Software Engineering and Advanced Applications*, pages 31–38, 2009.

[JG11] Wayne Jansen and Timothy Grance. Guidelines on security and privacy in public cloud computing. Technical report, National Institute of Standards and Technology, January 2011.

[jit] JBP - Java Byzantine Paxos. http://www.navigators.di.fc.ul.pt/software/jitt/jbp.html.

[JMM07] Flavio Junqueira, Yanhua Mao, and Keith Marzullo. Classic paxos vs fast paxos: Caveat emptor. In *Proc. of the Workshop on Hot Topics in System Dependability (HotDep'07)*, June 2007.

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

[KD04] Ramakrishna Kotla and Mike Dahlin. High-throughput Byzantine fault tolerance. In *Proceedings of the International Conference on Dependable Systems and Networks - DSN 2004*, June 2004.

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

[KSC$^+$10] Rüdiger Kapitza, Matthias Schunter, Christian Cachin, Klaus Stengel, and Tobias Distler. Storyboard: Optimistic deterministic multithreading. In *Proceedings of the 6th Workshop on Hot Topics in System Dependability – HotDep'10*, pages 1–8, October 2010.

[Lam86] Leslie Lamport. On interprocess communication (part II). *Distributed Computing*, 1(1):203–213, January 1986.

[LDLM09] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design & Implementation – NSDI'09*, April 2009.

[LKMS04] Jinyuan Li, Maxwell Krohn, David MaziËres, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proc. 6th Symp. Operating Systems Design and Implementation (OSDI)*, pages 121–136, 2004.

[LLSFV08] Jim Lau, Lau Cheuk Lung, Joni da S. Fraga, and Giuliana Santos Veronese. Designing fault tolerant web services using BPEL. In *Proc. of the Seventh IEEE/ACIS Intl. Conf. on Computer and Information Science*, pages 618–623, 2008.

[LR06] Barbara Liskov and Rodrigo Rodrigues. Tolerating Byzantine faulty clients in a quorum system. In *Proc. of the 26th IEEE Int. Conference on Distributed Computing Systems - ICDCS'06*, July 2006.

[LSP82a] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, July 1982.

[LSP82b] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, July 1982.

[MA06] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine consensus. *IEEE Trans. on Dependable and Secure Computing*, 3(3):202–215, July 2006.

[MAD02] Jean-Philippe Martin, Lorenzo Alvisi, and Mike Dahlin. Minimal Byzantine storage. In *Proc. of the 16th International Symposium on Distributed Computing - DISC 2002*, pages 311–325, October 2002.

[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/.

[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.

[MNCV06] Henrique Moniz, Nuno Ferreira Neves, Miguel Correia, and Paulo Verissimo. Randomized intrusion-tolerant asynchronous services. In *Proc. of the Int. Conf. on Dependable Systems and Networks – DSN'06*, jun 2006.

[MND+04] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39:21–41, 2004.

[MR98a] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, October 1998.

[MR98b] 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.

[MS02] David Mazières and Dennis Shasha. Building secure file systems out of Byzantine storage. In *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.

[MSL+10a] 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.

[MSL+10b] 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 '10)*, 2010.

[Nao09] Erica Naone. Are we safeguarding social data? Technology Review published by MIT Review, http://www.technologyreview.com/blog/editors/22924/, February 2009.

[NDO11] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs. In *Proc. of the Sixth EuroSys Conference*, pages 343–356, 2011.

[net] Netty - The Java NIO Client-Server Socket Framework. http://jboss.org/netty.

[NN00] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. *IEEE Journal on Selected Areas in Communications*, 18(4):561–570, April 2000.

[OBLC06] Rafael Rodrigues Obelheiro, Alysson Neves Bessani, Lau Cheuk Lung, and Miguel Correia. How practical are intrusion-tolerant distributed systems? DI-FCUL TR 06–15, Dep. of Informatics, Univ. of Lisbon, September 2006.

[ope] OpenStack. http://www.openstack.org.

[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.

[PTT08] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables. In *Proc. 15th ACM Conference on Computer and Communications Security (CCS)*, 2008.

[Rab89] Michael Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, February 1989.

[Sar09] David Sarno. Microsoft says lost sidekick data will be restored to users. *Los Angeles Times*, Oct. 15th 2009.

[SCC$^+$10a] 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.

[SCC$^+$10b] Alexander Shraer, Christian Cachin, Asaf Cidon, Idit Keidar, Yan Michalevsky, and Dani Shaket. Venus: Verification for untrusted cloud storage. In *Proc. Cloud Computing Security Workshop (CCSW)*. ACM, 2010.

[Sch90] Fred B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 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.

[SDM$^+$08] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. BFT protocols under fire. In *Proc. of 5th Symposium on Networked Systems Design and Implementation - NSDI 2008*, April 2008.

[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.

[SGR09] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. Towards trusted cloud computing. In *Proc. of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, June 2009.

[Sha79] Adi Shamir. How to share a secret. *Communications of ACM*, 22(11):612–613, November 1979.

[SK99] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information Systems*, 2:159–176, May 1999.

[sma] BFT-SMART - High-performance Byzantine fault-tolerant State Machine Replication. `http://code.google.com/p/bft-smart/`.

[SNV05] Paulo Sousa, Nuno Ferreira Neves, and Paulo Verissimo. How resilient are distributed f fault/intrusion-tolerant systems? In *Proceedings of Dependable Systems and Networks – DSN 05*, pages 98–107, June 2005.

[SSR$^+$10] Kunwadee Sripanidkulchai, Sambit Sahu, Yaoping Ruan, Anees Shaikh, and Chitra Dorai. Are clouds ready for large distributed applications? *SIGOPS Operating Systems Review*, 44:18–23, 2010.

[SvDO$^+$06] Luis F. G. Sarmenta, Marten van Dijk, Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas. Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In *Proceedings of the 1st ACM Workshop on Scalable Trusted Computing*, pages 27–42, November 2006.

[tcl10] Project TCLOUDS – trustworthy clouds - privacy and resilience for Internet-scale critical infrastructure. `http://www.tclouds-project.eu/`, 2010.

[Tou84] Sam Toueg. Randomized Byzantine agreements. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 163–178, 1984.

[TT05] Roberto Tamassia and Nikos Triandopoulos. Computational bounds on hierarchical data processing with applications to information security. In Luís Caires et al., editors, *Proc. 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3580 of *Lecture Notes in Computer Science*, pages 153–165. Springer, 2005.

[VCB$^+$09] Giuliana Santos Veronese, Miguel Correia, Alysson Bessani, Lau Chung, and Paulo Verissimo. Minimal Byzantine fault tolerance: Algorithm and evaluation. DI/FCUL TR 09–15, Department of Computer Science, University of Lisbon, June 2009.

[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 28th IEEE Symposium on Reliable Distributed Systems – SRDS'09*, September 2009.

[VNC03] Paulo Verissimo, Nuno Ferreira Neves, and Miguel Pupo Correia. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *LNCS*. 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.

[VNC08] Paulo Verissimo, Nuno Ferreira Neves, and Miguel Correia. The crutial reference critical information infrastructure architecture: A blueprint. *International Journal of System of Systems Engineering*, 1:78–95, 2008.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[VSV09] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. Cumulus: Filesystem backup to the cloud. *ACM Transactions on Storage*, 5(4):1–28, 2009.

[Vuk10] Marko Vukolic. The Byzantine empire in the intercloud. *ACM SIGACT News*, 41(3):105–111, 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 *Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation – OSDI 2006*, pages 307–320, 2006.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of the 18th ACM Symposium on Operating Systems Principles - SOSP'01*, 2001.

[Whi09] Tom White. *Hadoop: The Definitive Guide*. O'Reilly, 2009.

[WSS09] Peter Williams, Radu Sion, and Dennis Shasha. The blind stone tablet: Outsourcing durability to untrusted parties. In *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2009.

[WSV$^+$11] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. ZZ and the art of practical BFT execution. In *Proceedings of the 6th ACM SIGOPS/EuroSys European Systems Conference - EuroSys'11*, April 2011.

[WWL$^+$09] Qian Wang, Cong Wang, Jin Li, Kui Ren, and Wenjing Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *Proc. of the 14th European Symposium on Research Computer Security – ESORICS 09*, pages 355–370. Springer, 2009.

[YMV$^+$03] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement form execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles - SOSP'03*, pages 253–267, October 2003.

[Zie04] Piotr Zielinski. Paxos at war. Technical Report UCAM-CL-TR-593, University of Cambridge Computer Laboratory, Cambridge, UK, June 2004.