D2.2.4
Adaptive Cloud-of-Clouds Architecture, Services and Protocols

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 / D2.2.4 / 1.0
Activity and Work package contributing to deliverable:	Activity 2 / WP 2.2
Due date:	September 2013 – M36
Actual submission date:	30th September, 2013

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

Abstract:	This document describes a number of components and algorithms developed during the last year of the TClouds project, extending, complementing or superseding component descriptions in D2.2.1 and D2.2.2.
Keywords:	Resilience, adaptation, cloud-of-clouds, object storage, replication, Byzantine fault tolerance, state machine replication, database, file systems

Editor

Alysson Bessani (FFCUL)

Contributors

Alysson Bessani, Miguel Correia, Vinícius Cogo, Nuno Neves, Marcelo Pasin, Hans P. Reiser, Marcel Santos, João Sousa and Paulo Veríssimo (FFCUL)

Leucio Antonio Cutillo (POL)

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

Executive Summary

In this deliverable we describe our 3rd-year efforts in further developing replication tools for implementing dependable cloud-of-clouds services, with particular emphasis on techniques for replicated execution and its use for building dependable cloud services for the TClouds plataform. Most of the contributions here presented revolves around the BFT-SMART replication library, which implements the concept of Byzantine fault-tolerant state machine replication. Using this library, we show efficient techniques for implementing dependable durable services (i.e., services that use durable storage like magnetic disks or SSDs), adaptive services (services able to scale up and scale out) and also a transactional database based on an SQL DBMS. This report also describes C2FS, a cloud-of-clouds file system that makes use of the resilient object storage technology (extensively described in TClouds D2.2.2) and BFT-SMART for storing files in a secure and controlled way in a set of cloud storage services. The deliverable finishes with a brief discussion of a privacy-by-design P2P design that shows some promising ideas for future generations of cloud-of-clouds systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 TClouds — Trustworthy Clouds

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

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

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

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

1.2 Activity 2 — Trustworthy Internet-scale Computing Platform

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

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

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

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

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

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

1.3 Workpackage 2.2 — Cloud of Clouds Middleware for Adaptive Resilience

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

During the first year, a set of components and algorithms were identified, together with a reference architecture, and described in D2.2.1. The second year we focused on the fundamental protocols required for supporting cloud-of-clouds replication, namely asynchronous state machine replication and resilient object storage. These results, together with some other contributions, were presented in D2.2.2.

1.4 Deliverable 2.2.4 — Adaptive Cloud-of-Clouds Architecture, Services and Protocols

Overview. The increasing maturity of cloud computing technology is leading many organizations to migrate their IT solutions and/or infrastructures to operate completely or partially in the (public) cloud. Even governments and companies that maintain critical infrastructures are considering the use of public cloud offers to reduce their operational costs [Gre10]. Nevertheless, the use of public clouds (e.g., Amazon Web Services, Windows Azure, Rackspace) has limitations related to security and privacy, which should be accounted for, especially for the critical applications we are considering in TClouds.

These limitations are mostly related with the fact that currently cloud-based applications heavily depend on the trustworthiness of the cloud provider hosting the services. This trust may be unjustified due to a set of threats affecting cloud providers since their inception:

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

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

- **Loss and corruption of data**: There are several cases of cloud services losing or corrupting customer data. Two elucidative examples are: in October 2009 a subsidiary of Microsoft, Danger Inc., lost the contacts, notes, photos, etc. of a large number of users of the Sidekick service [Sar09]; in February of the same year Ma.gnolia lost half a terabyte of data that it never managed to recover [Nao09]. According to OSF' dataloss database[1], incidents involving data loss by third parties (including cloud providers) are still a serious threat.

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

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

One of the main objectives of WP2.2 is to develop techniques, proof-of-concept middleware and services that exploit the use of multiple cloud providers (the so called cloud-of-clouds) to mitigate these threats.

In this deliverable we present contributions related with the design and use of a replication library implementing the *State Machine Replication* component of TClouds. Contrary to the *Resilient Object Storage* (see D2.2.2), which assumes the replicas are simple storage services with read/write capabilities, the State Machine Replication component can be used to implement replicated execution of deterministic services [Sch90], being thus much more general and hard to implement.

In accordance with WP2.2 requirements (using public clouds to support dependable services), the replication library described in this deliverable (called BFT-SMaRt) does not require any special feature from replica' nodes or the communication network, being thus adequate to be used in any IaaS service providing VM instances (e.g., Amazon EC2), on the opposite of MinBFT [VCB+13], EBAWA [VCBL10] or CheapBFT [KBC+12] (the resource-efficient State Machine Replication of WP2.1) which require trusted components on the nodes.

Besides the design of the replication library, we also present extensive experimental work showing how this service supports durable services (i.e., services that make use of persistent storage like magnetic disks and SSDs), filling a gap in the distributed systems literature. In the same way, we deploy and analyze the performance of BFT-SMaRt in several wide-area deployments around Europe, simulating a cloud-of-clouds environment, to assess (1) if BFT-SMaRt (and state machine replication in general) is stable (i.e., its latency is predictable) and performant enough to be used in practice, and (2) if the optimization techniques proposed in several well-known papers do improve the performance BFT-SMaRt in wide-area setups. Finally, we also present a framework for dependable adaptation of replicated services that supports vertical and horizontal scalability, including experiments showing with BFT-SMaRt.

[1] See http://datalossdb.org/statistics?utf8=?&timeframe=all_time.

This deliverable also describe two storage systems developed using TClouds components. The first is SteelDB, a database replication middleware built over BFT-SMaRt that tolerates Byzantine faults and supports ACID transactions. The second is C2FS, a cloud-of-clouds file system that uses the Resilient Object Storage described in D2.2.2 (in particular, the DepSky system [BCQ+11]) to replicate files in set of cloud storage providers and BFT-SMaRt for implementing a cloud-of-clouds service for storing file metadata and coordinating accesses to shared files.

It is worth to mention that SteelDB and C2FS should *not* be considered core sub-systems of TClouds - in the same way as Resilient Object Storage (DepSky) and State Machine Replication (BFT-SMaRt) are - but *experimental services* developed using the TClouds enabling middleware for easy integration with the project application scenarios. SteelDB is used in the demonstration of the Smart Lighting scenario (WP3.2) while C2FS is used in the Healthcare scenario (WP3.1).

Since TClouds is a research project, we also present a novel proposal for a privacy-by-design cloud-of-clouds architecture. In such architecture, cloud services are provided by a number of cooperating independent parties consisting in the user nodes themselves. Unlike the current cloud services, the proposed solution provides user anonymity and untraceability, and allows users to select service providers on the basis of the expected privacy protection. In a scenario where each user runs the TClouds platform on his own infrastructure, such architecture sheds new lights on privacy preserving trustworthy cloud computing.

Structure. This deliverable is organized in the following way. Chapter 2 offers a extended description of BFT-SMaRt, the WP2.2 implementation of the State Machine Replication component for deployment in IaaS. This chapter supersedes and updates Chapter 8 of TClouds D2.2.1. The next chapter (3) discusses the problems of supporting durable services using State Machine Replication and present solutions for these problems. The material on this chapter was published in [BSF+13]. Chapter 4 shows some evaluation of the BFT-SMaRt replication library when the replicas are deployed around Europe, in a wide-area network. The material on this chapter will be presented on the 1st Workshop on Planetary-Scale Distributed Systems, in September 30th, 2013 (it is a no proceedings workshop). All the software related with these three chapters was delivered in D2.2.3 and is also released in as open source under the Apache License [BSA].

Chapter 5 presents the FITCH framework for supporting adaptive replicated services, including the BFT-SMaRt replication library. The material on this chapter was published in [CNS+13]. The next two chapters describe two storage systems developed using BFT-SMaRt. Chapter 6 describes Byzantine fault-tolerant transactional SQL database while Chapter 7 presents C2FS, the TClouds' Cloud-of-clouds file system. These five chapters provide extensive practical insights (including a lot of experimental work) on how to build and deploy SMR-based services in the real world.

Finally, Chapter 8 proposes a new Cloud-of-Clouds architecture which addresses the protection of the user's privacy from the outset and has been published in [CL13].

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

Target Audience. This deliverable aims at researchers and developers of security and management systems for cloud-computing platforms. The deliverable assumes graduate-level background knowledge in computer science technology, specifically, in operating system concepts

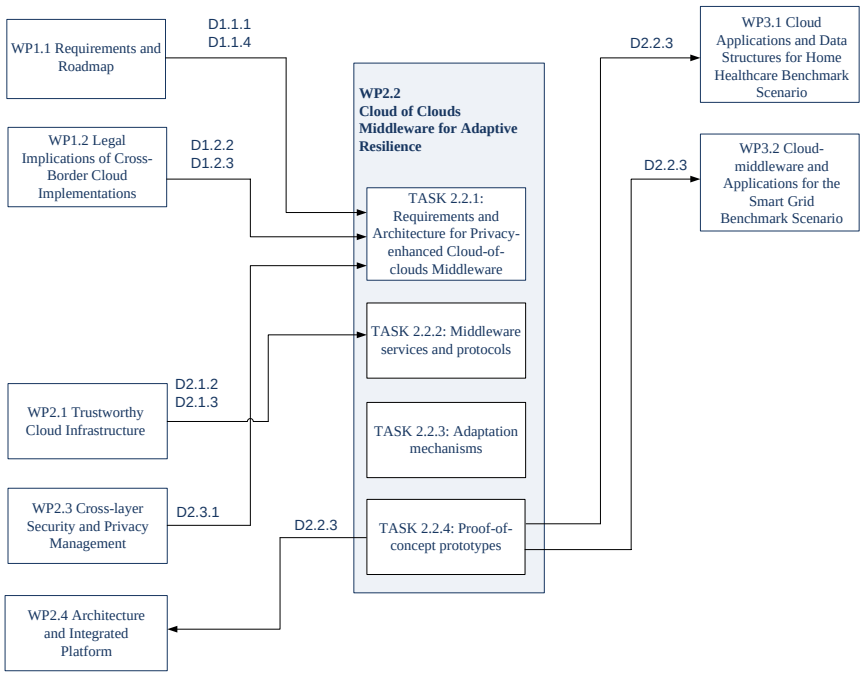


Figure 1.1: Graphical structure of WP2.2 and relations to other workpackages.

and distributed systems models and technologies.

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

The present deliverable, D2.2.4, presents a set of contributions related with the use of Byzantine fault-tolerant replication to implement trustworthy services and applications. This deliverable extends and uses the contributions described in two previous WP2.2 deliverables (D2.2.1 and D2.2.2) and defines a subset of services that were demonstrated in TClouds 2nd-year and 3rd-year demos. Naturally, most of the ideas presented here will be used in the components used in both use cases of A3.

Chapter 2

State Machine Replication (BFT-SMaRt) Revisited

Chapter Authors:
Alysson Bessani and João Sousa (FFCUL).
This chapter updates Chapter 8 of TClouds D2.2.1.

2.1 Introduction

The last decade and half saw an impressive amount of papers on Byzantine Fault-Tolerant (BFT) State Machine Replication (SMR) (e.g., [CL02, CWA+09, 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 are no robust-enough implementations of BFT SMR available, only prototypes used as proof-of-concept for the papers, makes it quite difficult to use this kind of technique. The general perception is that implementing BFT protocols is far from trivial and that commission faults are rare and can be normally dealt with simpler techniques like checksums,

To the best of our knowledge, from all "BFT systems" that appeared on the last decade, only the early PBFT [CL02] and the very recent UpRight [CKL+09] implement an almost complete replication system (which deal with the normal synchronous fault-free case and the 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, besides having too low performance (at least the implementation currently available) from what is currently accepted as the state-of-the-art.

In this chapter we describe our effort in implementing and maintaining BFT-SMaRt [BSA], 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, including associate protocols like state transfer [BSF+13] and reconfiguration. In particular, BFT-SMaRt is the first BFT SMR system to fully support reconfiguration of the replica set.

The chapter is organized as follows: §2.2 describes the design of BFT-SMaRt. The library's architecture is presented in §2.3. §2.5 discusses the results we obtained from our micro-benchmarks and experiments. §2.6 highlight some lessons learned during these 6 years of development and maintenance of our system. §2.7 presents our concluding remarks.

2.2 BFT-SMaRt Design

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]. In 2009 we revamped the design of this multicast library to make it a complete BFT replication library, including features such as checkpoints and state transfer. Nonetheless, it was only during the TClouds project (2010-2013) that we could improve the system to go much further than any other open-source state machine replication (BFT or not) we are aware of in terms of quality and functionality.

2.2.1 Design Principles

BFT-SMaRt was developed with the following design principles in mind:

- **Tunable fault model.** By default, BFT-SMaRt tolerates non-malicious Byzantine faults, which is a realistic system model in which replicas can crash, delay messages or corrupt its state, just like what is observed in many real systems and components. We think this is the most appropriate system model for a pragmatical system that aims to serve as a basis for implementing critical services. Besides that, BFT-SMaRt also support the use of cryptographic signatures for improved tolerance to *malicious Byzantine faults*, or the use of a simplified protocol, similar to Paxos [Lam98], to tolerate only crashes and message corruptions[1].

- **Simplicity.** The emphasis on protocol correctness made us avoid the use of optimizations that could bring extra complexity both in terms of deployment and coding or add corner cases to the system. For this reason, we avoid techniques that, although promising in terms of performance (e.g., speculation [KAD+07b] and pipelining [GKQV10]) or resource efficiency (e.g., trusted components [VCB+13] or IP multicast [CL02,KAD+07b]), would make or code more complex to make correct (due to new corner cases) or deploy (due to lack of infrastructure support). This emphasis also made us chose Java instead of C/C++ as the implementation language. Somewhat surprisingly, even with these design choices, the performance of our system is still better than some of these optimized SMR implementations.

- **Modularity.** BFT-SMaRt implements the Mod-SMaRt protocol [SB12] (also described in Chapter 6 of TClouds D2.2.2), a *modular* SMR protocol that uses a well defined consensus module in its core. On the other hand, system like PBFT implements a *monolithic* protocol where the consensus algorithm is embedded inside of the SMR, without a clear separation. While both protocols are equivalent at run-time, modular alternatives tend to be easier to implement and reason about, when compared to monolithic protocols. Besides the existence of modules for reliable communication, client requests ordering and consensus, BFT-SMaRt also implements state transfer and reconfiguration modules, which are completely separated from the agreement protocol, as show in Figure 2.1.

- **Simple and Extensible API.** Our library encapsulates all the complexity of SMR inside a simple and extensible API that can be used by programmers to implement deterministic

[1] During this chapter we only consider the BFT setup of the system, we will discuss the crash fault tolerance simplifications later in §2.4.

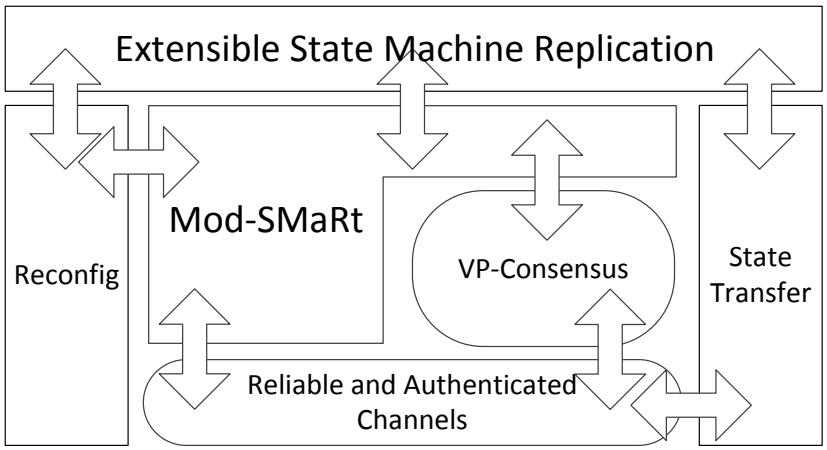


Figure 2.1: The modularity of BFT-SMART.

services. More precisely, if the service strictly follows the SMR programming model, clients can use a simple *invoke(command)* method to send commands to the replicas, that implement an *execute(command)* method to process the command, after it is totally ordered by the framework. If the application requires advanced features not supported by such basic programming model, these features can be implemented with a set or alternative calls (e.g., *invokeUnordered*), callbacks or plug-ins both at client and server-side (e.g., custom voting by the client, reply management and state management, among others).

- **Multi-core awareness.** BFT-SMART takes advantage of ubiquitous multicore architecture of servers to improve some costly processing tasks on the critical path of the protocol. In particular, we make our system throughput scale with the number of hardware threads supported by the replicas, specially when signatures are enabled.

2.2.2 System Model

BFT-SMART assumes a set of n replicas and an unknown number of clients. Up to $f < n/3$ replicas and an unbounded number of clients can fail arbitrarily. Moreover, we assume faulty replicas can crash and later recover and that replicas can be added or removed from the system.

BFT-SMART requires an eventually synchronous system model like other protocols for SMR for ensuring liveness [CL02, Lam98]. Moreover, we assume reliable authenticated point-to-point links between processes. These links are implemented using message authentication codes (MACs) over TCP/IP. The system currently support two implementations for this: either the keys are defined statically, one for each pair of processes, in configuration files on the clients and replicas, or a public-key infrastructure is used to support for the Signed Diffie-Hellman for generating these keys when connections are established.

2.2.3 Core Protocols

BFT-SMART uses a number of protocols for implementing state machine replication. In this section we give a brief overview of these protocols.

Atomic Broadcast

Atomic broadcast is achieved using the Mod-SMaRt protocol [SB12] together with the Byzantine consensus algorithm described in [Cac09]. Clients send their requests to all replicas, and wait for their replies. In the absence of faults and presence of synchrony, BFT-SMART executes in *normal phane*, whose message pattern is illustrated in Figure 2.2. This is comprised by a sequence of consensus executions. Each 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 STATE message. All replicas that receive this message verify if its sender is the current leader, and if the value proposed is valid, they weakly accept the value being proposed, sending an WRITE message to all other replicas. If some replica receives more than $\frac{n+f}{2}$ WRITE messages for the same value, it strongly accepts this value and sends a ACCEPT message to all other replicas. If some replica receives more than $\frac{n+f}{2}$ ACCEPT messages for the same value, this value is used as the decision for consensus. Such decision is logged (either in memory or disk), and the requests it contains are executed.

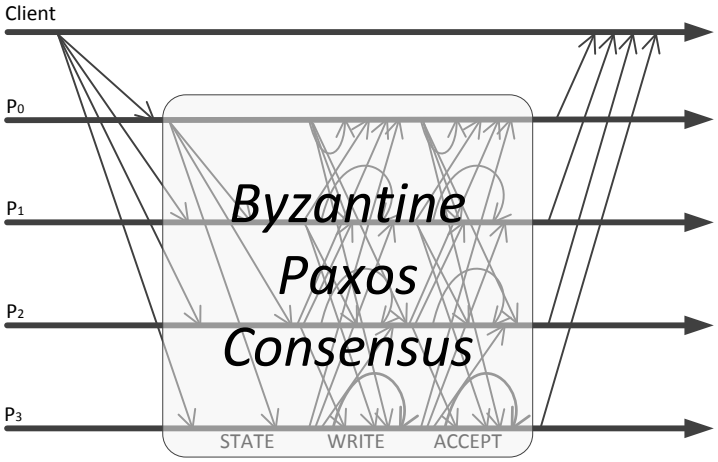


Figure 2.2: BFT-SMART Message pattern during normal phase.

The normal phase of the protocol is executed in the absence of faults and in the presence of synchrony. When these conditions are not satisfied, the *synchronization phase* might be triggered. During this phase, Mod-SMaRt must ensure three things: (1) a quorum of $n - f$ replicas must have the pending messages that caused the timeouts; (2) correct replicas must exchange logs to converge to the same consensus instance; and (3) a timeout is triggered in this consensus, proposing the same leader at all correct replicas (see [SB12] for details).

State Transfer

In order to implement a practical state machine replication, the replicas should be able to be repaired and reintegrated in the system, without restarting the whole replicated service. Mod-SMaRt by itself can guarantee total ordering of operations across correct replicas, but it does not account for replicas that are faulty but eventually recover.

In order for replicas to be able to transfer the application's state to another replica, it first needs to fetch this state from the service. But fetching it each time an operation is executed degrades performance, especially if the state is large. One could make use of periodic snapshots, called checkpoints, and logs of operations to avoid fetching the state all the time. But on the other hand, applications that use BFT-SMART might already have this functionality (such as

the case of database management systems - see Chapter 6). Because of this, BFT-SMART delegates to the application the task of managing checkpoints, thus enabling state management to be as flexible as possible (even non-existent, if the application does not need it).

The default state transfer protocol can be triggered either when (1) a replica crashes but later is restarted, (2) a replica detects that it is slower that the others, (3) a synchronization phase is triggered but the log is truncated beyond the point at which the replica could apply the operations, and (4) the replica is added to the system while already executing system (see next section). When any of these scenarios are detected, the replica sends a *STATE_REQUEST* message to all the other replicas asking for the application's state. Upon receiving this request, they reply with a *STATE_REPLY* message containing the version of the state that was requested by the replica. Instead of having one replica sending the complete state (checkpoint and log) and others sending cryptographic hashes for validating this state, like is done in PBFT and all other systems we are aware of, we use a partitioning scheme in which one replica send a checkpoint and the others send parts of the logs (see [BSF+13] for details).

Reconfiguration

All previous BFT SMR systems assume a static system that cannot grow or shrink over time. BFT-SMART, on the other hand, provides and additional protocol that enables the system's view to be modified during execution time, i.e., replicas can be added or removed from the system without needing to halt it. In order to accomplish this, BFT-SMART uses a special type of client named *View Manager*, which is a trust party. The View Manager is expected to be trustworthy, and managed only by system administrators. It can also remain off-line for most of the time, being required only for adding or removing replicas.

The reconfiguration protocol works as follows: the View Manager issues a special type of operation which is submitted to the Mod-SMaRt algorithm as any other operation, in a similar way as described in [Lam98, LMZ10a]. Through these operations, the View Manager notifies the system about the IP address, port and id of the replica it wants to add to (or remove from) the system. Because these operations are also totally ordered, all correct replicas will adopt the same system's view.

Once the View Manager operation is ordered, it is not delivered to the application. Instead, the information regarding the replicas present in the system is updated, and the parameter f is set in function of the updated list of replicas that comprise the system. Moreover, the replicas start trying to establish a secure channel with this new replica. Following this, the replicas reply to the View Manager informing it if the view change succeeded. If so, the View Manager sends a special message to the replica that is waiting to be added to (or removed from) the system, informing it that it can either start executing or halt its execution. After this point, if a replica is being added, it triggers the state transfer protocol to bring itself up to date.

2.3 Implementation Choices

The codebase of BFT-SMART contains less than 13.5K lines of commented Java code distributed in little more than 90 classes and interfaces. This is much less than what was used in similar systems: PBFT contains 20K lines of C code and UpRight contains 22K lines of Java code.

2.3.1 Building blocks

To achieve modularity, 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 three groups: *communication system*, *state machine replication* and *state management*. The first 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 while the second implements the core algorithms for establishing total order of requests. The third implements the state management, which is described in [BSF+13], and thus omitted here.

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. The main three modules are:

- **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 [JBo10] for implementing client/server communication. The most important requirement of this module is that it should be able to accept and deal with few thousands 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 connected 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 six 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. This thread also triggers the broadcast of the STATE message to the other replicas within the system.

- **Acceptor**: this module implements the core of the consensus algorithm: messages of type ACCEPT and WRITE are processed and generated here. For instance, when more than $\frac{n+f}{2}$ ACCEPT messages for the same (proposed) value are received, a corresponding WRITE message is generated.

- **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.

- **Leader Change Manager**: Most of the complex code to deal with leader changes is in this module.

- **Reconfiguration Manager**: The reconfiguration protocol is implemented by this module.

2.3.2 Staged Message Processing

A key point when implementing a high-throughput replication middleware 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 behaviors from both replicas and clients.

Figure 2.3 presents the main architecture with the threads used for staged message processing [WCB01] 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.

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 client' MACs and signatures (optionally supported) 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 next consensus; (ii) the previous instance 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 a batch of new requests to be ordered as soon as a previous consensus is decided and there are pending messages from clients. Notice the size will contain all pending requests (up to a certain maximum size, defined in the configuration file), so there is no waiting to fill a batch of certain size before proposing. In non-leader replicas, this thread is always sleeping waiting for condition (i).

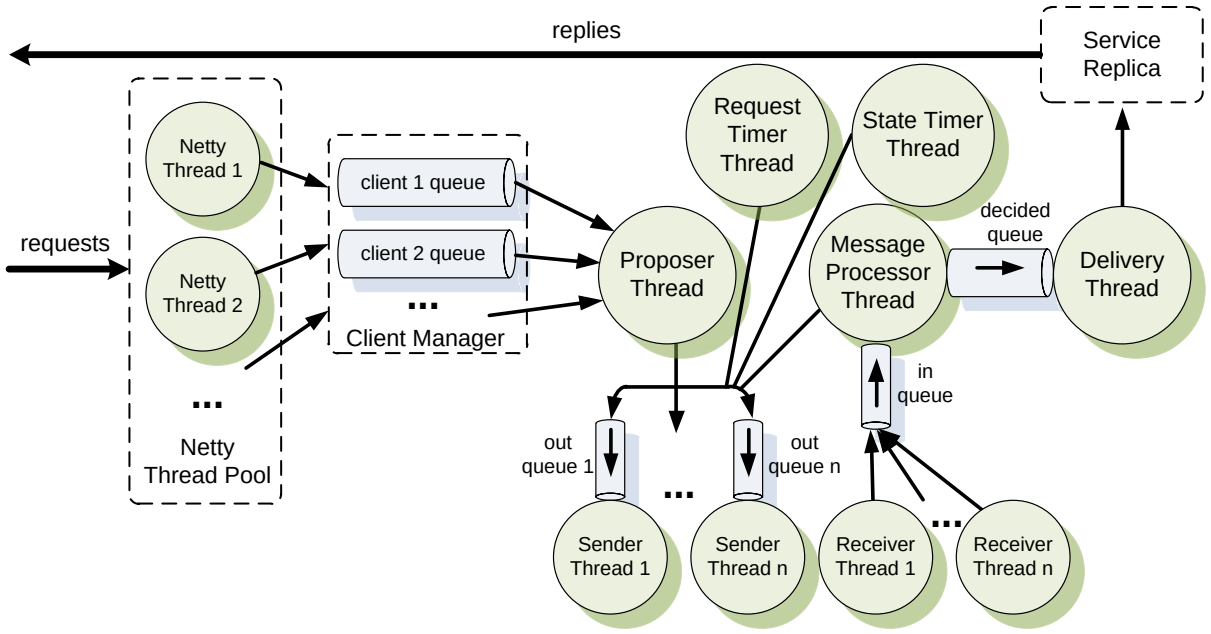


Figure 2.3: The staged message processing on BFT-SMaRt.

Every message m to be sent by one replica to another 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. As a side note, it is worth to mention that although the STATE message contains the whole batch of messages to be ordered, the ACCEPT and WRITE messages only contain the cryptographic hash of this batch.

When a consensus is finished on a replica (i.e., the replica received more than $\frac{n+f}{2}$ WRITEs 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, a state management thread stores the batch of requests in the log. In parallel with that, the delivery thread invokes the service replica to make it execute the request and generate a reply. When the batch is properly logged and the response is generated by the replica, the reply is sent to the invoking clients.

The *request timer task* is periodically activated to verify if some request remained 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.

The *state timer thread* is responsible for managing the timeout associated with the state transfer protocol. This timeout is necessary because the state transfer protocol requires the cooperation of all active; if one of such replica is faulty, it might not send the state, thus freezing the current state transfer. If this timeout expires, the state transfer protocol is re-initialized, and a different replicas is chosen to send the entire state.

Although we do not claim that the architecture depicted in Figure 2.3 is the best architecture for implementing state machine replication, it is the result of several experiences to make the most simple and performant SMR implementation in Java.

2.4 Alternative Configurations

As already mentioned in previous sections, by default BFT-SMaRt tolerates non-malicious Byzantine faults, in the same way as most work on BFT replication (e.g., [CL02, KAD+07b]). However, the system can be tuned or deployed in environments to tolerate only crashes or (intelligent) malicious behavior.

2.4.1 Simplifications for Crash Fault Tolerance

BFT-SMaRt supports a configuration parameter that, if activated, makes the system strictly crash fault-tolerant. When this feature is active, the system tolerates $f < n/2$ (i.e., it requires only a majority of correct replicas), which requires modification in all required quorums of the protocols, and bypasses one *all-to-all* communication step during the consensus execution (the ACCEPT round of the consensus is not required). Other than that, the protocol is the same as in the BFT case, with MACs still enabled for message verification, bringing also tolerance to message corruption.

2.4.2 Tolerating Malicious Faults (Intrusions)

Making a BFT replication library tolerate intrusions requires one to deal with several concerns that are not usually addressed by most BFT protocols [Bes11]. In this section we discuss how BFT-SMaRt deal with some of these concerns.

Previous works showed that the use of public key signatures on client requests makes it impossible for clients to forge MAC vectors and force leader changes (making the protocol much more resilient against malicious faults) [ACKL08, CWA+09]. By default, BFT-SMaRt does not use public-key signatures other than for establishing shared symmetric keys between replicas, however the system optionally support the use of signatures for avoiding this problem.

The same works also showed that a malicious leader can launch undetectable performance degradation attacks, making the throughput of the system as small as 10% of what would be achieved in fault-free executions. Currently, BFT-SMaRt does not provide defenses against such attacks, however, the system can be easily extended to support periodic leader changes to limit such attacks damage [CWA+09]. In fact, the codebase of a very early version of BFT-SMaRt were already used to implement a protocol resilient against to this kind of attack [VCBL09].

Finally, the fact we developed BFT-SMaRt in Java make it easy to deploy it in different platforms[2] for avoiding single mode failures, being caused by accidental events (e.g., a bug

[2]Although we did not support N-versions of the system codebase, we believe supporting the deployment in several platforms is a good compromise solution.

or infrastructure bug) and malicious attacks (e.g., caused by a common vulnerabilities). Such platforms comprise the deployment of replicas in different operating systems [GBG+13] or even cloud providers [Vuk10]. In the latter case, our protocol need to be prepared for dealing with wide-area replication. The behavior of the BFT-SMaRt in WANs is studied in Chapter 4 and some experiments with a real deployment in a cloud-of-clouds is described in §7.6.2.

2.5 Evaluation

In this section we present results from BFT-SMaRt's performance evaluation. These experiments consist of (1) some micro-benchmarks designed to evaluate the library's throughput and client latency and (2) an experiment designed to depict the performance's evolution of a small application implemented with BFT-SMaRt once the system is forced to withstand events like replicas faults, state transfers, and system reconfigurations.

2.5.1 Experimental Setup

All experiments ran with three to five replicas hosted in separated machines. The client processes were distributed uniformly across another four machines. Each client machine ran up to eight Java processes, which in turn executed up to fifty threads which implemented BFT-SMaRt clients (for a total of up to 1600 client processes).

Clients and replicas executed in the Java Runtime environment 1.7.0_21 on Ubuntu 10.04, hosted in Dell PowerEdge R410 servers. Each machine has two quad-core 2.27 GHz Intel Xeon E5520 processor with hyperthreading and 32 GB of memory. All machines communicate through a gigabit Ethernet switch isolated from the rest of the network.

2.5.2 Micro-benchmarks

PBFT-like Benchmarks. We start by reporting the results we gathered from a set of micro-benchmarks that are commonly used to evaluate state machine replication systems, and focus on replica throughput and client latency. They consist of a simple client/service implemented over BFT-SMaRt that performs throughput calculations at server side and latency measurements at client side. Throughout results were obtained from the leader replica, and latency results from a selected client.

Figure 2.4 illustrates BFT-SMaRt performance in terms of client latency against replica throughput for both BFT and CFT protocols. For each protocol we executed four experiments for different request/reply sizes: 0/0, 100/100, 1024/1024 and 4096/4096 bytes. Figure 2.4 shows that for each payload size, the CFT protocol consistently outperforms its BFT counterpart. Furthermore, as the payload size increases, BFT-SMaRt overall performance decreases. This is because (1) the overhead of requests/replies transmission between clients and replicas increases with the messages size, and (2) since Mod-SMaRt orders requests in batches, the larger is the payload, the bigger (in bytes) the batch becomes, thus increasing its transmission overhead amongst replicas.

We complement the previous results with Table 2.1, which shows how different payload's combinations affect throughput. This experiment was conducted under a saturated system running 1600 clients using only the BFT protocol. Our results indicate that increasing request's payload generates greater throughput degradation than reply's payload does. This can also be

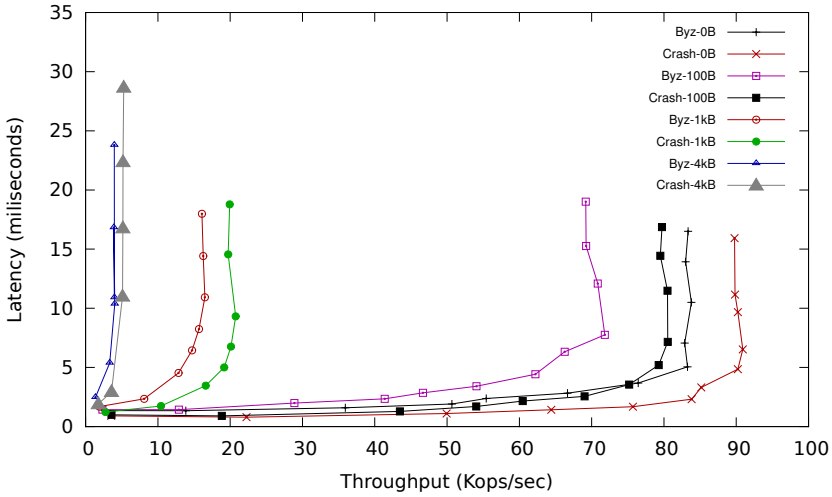


Figure 2.4: Latency vs. throughput configured for $f = 1$.

Requests \ Replies	0 bytes	100 bytes	1024 bytes
0 bytes	83337	75138	37320
100 bytes	78711	72879	36948
1024 bytes	16309	16284	15878

Table 2.1: Throughput for different requests/replies sizes for $f = 1$. Results are given in operations per second.

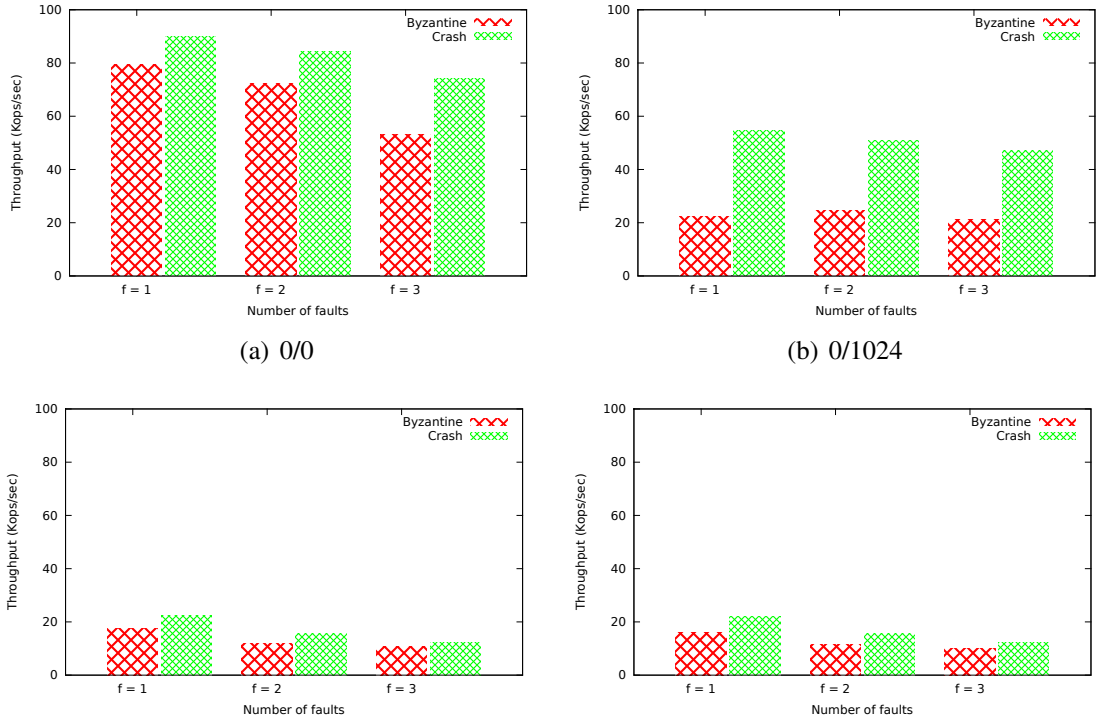
explained by the larger batch submitted to the consensus protocol, since request's payload influences its size, whereas reply's payload does not.

Fault-scalability. Our next experiment consider the impact of the size of the replica group on the peak throughput of the system for the system under different benchmarks. The results are reported in Figure 2.5.

The results show that, for all benchmarks, the performance of BFT-SMART degrades graciously as f increases, both for CFT and BFT setups. In principle, these results contradict the observation that protocols containing all-to-all communication patters are less scalable as the number of faults tolerated [AEMGG+05a]. However, this is not the case in BFT-SMART because (i) it exploit the cores of the replicas (which our machines have plenty) to calculate MACs, (i) only the $n-1$ propose message of the agreement protocol is significantly bigger, the other $(n-1)^2$ are much smaller and contain only a hash of the proposal, and (i) we avoid the use of multicast IP, which is know to cause problems with many senders (e.g., multicast storms).

Finally, it is also interesting to see that, with relatively big requests (1024 bytes), the difference between BFT and CFT tend to be very small, independently on the number of tolerated faults. Moreover, the performance drop between tolerating 1 to 3 faults is also much smaller with big payloads (both requests and replies).

Mixed workloads. Our next experiment considers a mix of requests representing reads and writes. In the context of this experiment, the difference between reads and writes is that the former issues small requests (almost-zero size) but gets replies with payload, whereas the latter issues requests with payload but gets replies with almost zero size. This experiment was also



(a) 0/0 (b) 0/1024

(c) 1024/0 (d) 1024/1024

Figure 2.5: Peak throughput of BFT-SMaRt for CFT ($2f + 1$ replicas) and BFT ($3f + 1$ replicas) considering different workloads and number of tolerated faults.

conducted under a saturated system running 1600 clients.

We performed the experiment both for the BFT and CFT setups of BFT-SMaRt, using requests/replies with payloads of 100 and 1024 bytes. Similarly to the previous experiments, the CFT protocol outperforms its BFT counterpart regardless of the ratio of read to write requests by around 5 to 15%. However, the observed behavior of the system regarding the throughput differs between the case of 100 bytes and 1024 payloads, whereas the former clearly benefits from a larger read/write ratio.

This happens because 1024 bytes requests (a write operation) generate batches much larger than requests with only 100 bytes of payload. This in turn spawns a much greater communication overhead in the consensus protocol. Therefore, as we increase the read to write ratio for payloads of 1024 bytes, the consensus overhead decreases, which in turn improves performance. This happens with up to 75% reads, which has a better throughput than 95%- or 100%-read workloads. This happens for payloads of 1024 bytes because at this point sending the large replies of the read become the contention point of our system. Notice this behavior is much less significant with small payloads.

Signatures and Multi-core Awareness. Our next experiment considers the performance of the system when signatures are enabled, and used for ensuring resilience to malicious clients [CWA+09]. In this setup a client sign every request to the replicas that first verify its authentic before ordering it. There are two fundamental service-throughput overheads involved in using 1024-bit RSA signatures. First, the messages are 112 bytes bigger than when SHA-1 MACs are used. Second, the replicas need to verify the signatures, which is relatively costly computational operation.

Figure 2.7 shows the throughput of BFT-SMaRt with different number of threads being

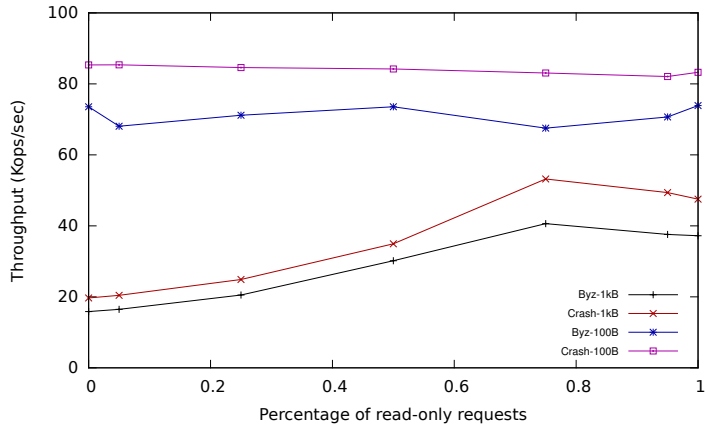


Figure 2.6: Throughput of a saturated system as the ratio of reads to writes increases for $n = 4$ (BFT) and $n = 3$ (CFT).

used for verifying signatures. As the results show, the architecture of BFT-SMaRt exploit the existence of multiple cores (or multiple hardware threads) to scale the throughput of the system. This happens because the signatures are verified by the Netty thread pool, which uses a number of threads proportional to the number of hardware threads in the machine (see Figure 2.3).

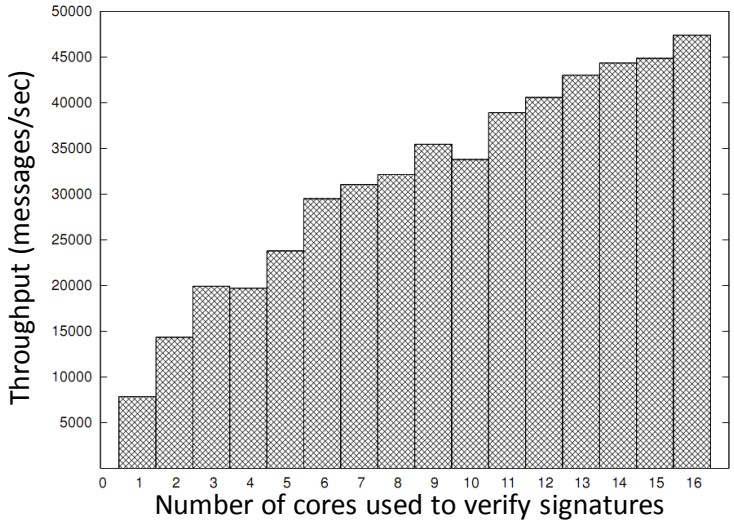


Figure 2.7: Throughput of BFT-SMaRt using 1024-bit RSA signatures for 0/0 payload and $n = 4$ considering different number of hardware threads.

BFT-SMaRt vs. Others. As mentioned before, BFT-SMaRt was built for correctness, modularity and completeness, avoiding many of the optimizations that make SMR systems so complex to implement. Nonetheless, the overall performance of the system is better than competing (less robust) prototypes found on the Internet. Table 2.2 compares the *peak sustained throughput* of BFT-SMaRt, PBFT [CL02] and libPaxos (a crash fault-tolerant replication library [Pri]) in our environment. Notice that both PBFT and libPaxos were implemented in C.

System	*Throu.* (Kops/sec)	*Clients*
BFT-SMaRt	83	1000
BFT-SMaRt (crash)	91	800
PBFT	49	60
libPaxos	63	50

Table 2.2: Throughput of different replication systems for the 0/0 workload and the number of clients required to reach these values.

2.5.3 BFTMapList

In this section we discuss the implementation and evaluation of BFTMapList, a replicated in-memory data structure commonly used in social network applications. This experiment was designed to evaluate the behavior of an application implemented using BFT-SMaRt, and how it fares against replica's failures, recoveries, and reconfigurations.

Design and implementation. BFTMapList is an implementation of the *Map* interface from the Java API which uses BFT-SMaRt to replicate its data in set of replicas. It can be initialized at client side providing transparency of this underlying replication mechanism. This is done by invoking BFT-SMaRt within its implementation. In BFTMapList, keys correspond to string objects and values correspond to a list of strings. We implemented the *put*, *remove*, *size* and *containsKey* methods of the aforementioned Java interface. These methods insert/delete a new String/List pair, retrieve the amount of values stored, and check if a given key was already inserted in the data structure. We also implemented an additional method called *putEntry* so that we could directly add new elements to the lists given their associated key.

To evaluate the implementation of this data structure, we created client threads that constantly insert new strings of 100 bytes to these lists, but periodically purge them to prevent the lists from growing too large and exhaust memory. Each client thread corresponds to one BFT-SMaRt client. Besides storing data, the server side implementation also performs throughput calculations.

Results. We sought to observe how BFTMapList performance would evolve upon several events within the system - ranging from replicas faults, leader changes, state transfers and system reconfigurations. For this experiment, BFT-SMaRt were configured to tolerate Byzantine faults. Our results are depicted in Figure 2.8. We launched 30 clients issuing the *put*, *remove*, *size* and *putEntry* methods described previously over the course of 10 minutes. We started the experiment with an initial set of replicas with IDs ranging from 0 to 3. The throughput depicted in the figure is collected from replica 1.

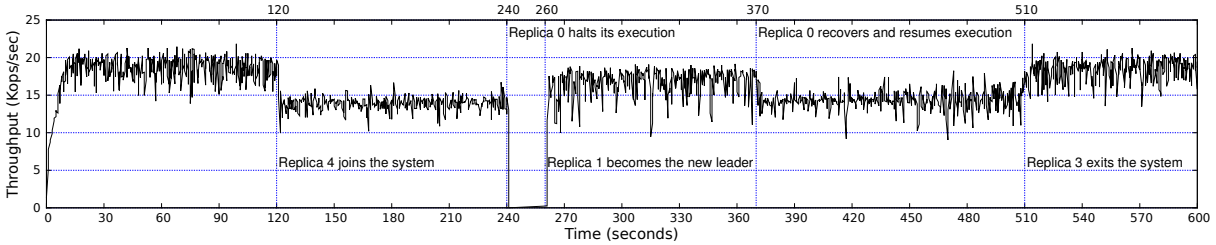


Figure 2.8: Throughput evolution across time and events, for $f = 1$.

As the clients started their execution, the service's throughput increased until all clients were operational around second 10. At second 120 we inserted replica 4 into the service. As

we did this, we observed a decreased in throughput. This can be explained by the fact that more replicas demand larger quorums in the consensus protocol and more messages to be processed in each replica. This reconfiguration spawns more message exchanges among replicas, which add congestion to the network and results in inferior performance.

At second 240, we crashed replica 0 (the current consensus' leader). As expected, the throughput dropped to zero during the 20 seconds (timeout value) that took the remaining replicas to trigger their timeouts and run Mod-SMaRt's synchronization phase. After this phase was finished, the system resumed execution. Since at this point there are less replicas executing, there are also less messages being exchanged in the system and the throughput was similar the initial configuration.

At second 370, we restarted replica 0, which resumes normal operation after triggering the state transfer. Upon its recovery, the system goes back to the throughput exhibited before replica 0 had crashed.

At second 510, we removed replica 3 from the group, thus setting the quorum size to its original value, albeit with a different set of replicas. Since there is one less replica to handle messages from, we are able to observe the system's original throughput again by the end of the experiment.

2.6 Lessons Learned

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

2.6.1 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 was specially important for client requests that are put in large quantities on batches to be decided by a consensus[3].

[3] A serialized 0-byte operation request requires 134 bytes with Java default serialization and 22 bytes in our custom serialization.

2.6.2 How to test BFT systems?

Although distributed systems tracing and debugging is a lively research area (e.g., [SBG10, BKSS11]), there are still no tools mature enough to be used. Our approach for testing BFT-SMART is based on the use of JUnit, a popular unity testing tool. In our case we use it in the final automatic test of our build script to run test scripts that (1.) setup replicas, (2.) run some client accessing the replicated service under test and verify if the results are correct, and (3.) kill the replicas in the end. Notice this is a completely black-box testing, the only way to observe the system behavior is through the client. Similar approaches are being used in other distributed computing open-source projects like Apache Zookeeper (which also uses JUnit).

Our JUnit-test framework allow us to easily inject crash-faults on the replicas, however, testing the system against malicious behaviors is much more tricky. The first challenge is to identify the critical malicious behaviors that should be injected on up to f replicas. The second challenge is how to inject the code of the malicious behaviors on these replicas. The first challenge can only be addressed with careful analysis of the protocol being implemented. Disruptive 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). Our pragmatic test approach can be complemented with orthogonal methods such as the Netflix chaos monkey [BT12] to test the system on site.

It is worth to notice that most faulty behaviors can cause bugs that affect the liveness of the protocol, since basic invariants implemented in key parts of the code can ensure safety (e.g., a leader proposing different values to different replicas should cause a leader change, not a disagreement). This means that a lot of recent work in verification of safety properties in distributed systems through model checking (e.g., [BKSS11]) does not solve the most difficult problem in the design of distributed protocols: liveness bugs.

Moreover, the fact that the system tolerates arbitrary faults makes it mask some non-deterministic bugs, or Heisenbugs, turning the whole test process even more difficult. For example, an older version of the BFT-SMART communication system loosed some messages sporadically when under heavy load. The effect of this was that in certain rare conditions (e.g., when the bug happens in more than f replicas during the same protocol phase) there was a leader change, and the system blocks. We call these bugs *Byzenbugs*, since they are a specific kind of Heisenbugs that happen in BFT systems and that only manifest themselves if they occur in more than f replicas at the same time. Consequently, these bugs are orders of magnitude more difficult to discover (they are masked) and very complex to reproduce (they seldom happen).

2.6.3 Dealing with heavy loads

When testing BFT-SMART under heavy loads, we found several interesting behaviors 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 (which runs more often that expected).

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 2.2 does not happen in practice. This same behavior 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 behavior that commonly happens in several distributed systems is that their throughput tends to drop after some time under heavy-load. This behavior is called *trashing* and can be avoided through a careful selection of the data structures[4] used on the protocol implementation and bounding the queues used for threads communication.

2.6.4 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 2007, 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 §2.5: a 1024-bit RSA signature creation went from 15 ms to less than 1.7 ms while its verification went from 1 ms to less than 0.09 ms (a $10\times$ improvement). 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).

2.7 Conclusions

This chapter reported our effort in building the BFT-SMART state machine replication library. Our contribution with this work is to fill a gap in SMR/BFT literature describing how this kind of protocol can be implemented in a safe and performant way. Our experiments show, the current implementation already provides a very good throughput for both small- and medium-size messages.

The BFT-SMART system described here is available as open source software in the project homepage [BSA] and, at the time of this writing, there are several groups around the world currently using or modifying our system for their research needs. In the next five chapters we explore other works and services based on BFT-SMART within the context of the TClouds project.

[4]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.

Chapter 3

Improving the Efficiency of Durable State Machine Replication

Chapter Authors:
Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, Miguel Correia (FFCUL)

3.1 Introduction

Internet-scale infrastructures rely on services that are replicated in a group of servers to guarantee availability and integrity despite the occurrence of faults. One of the key techniques for implementing replication is the *Paxos* protocol [Lam98], or more generically the *State Machine Replication* (SMR) approach [Sch90], which is one of the core sub-systems of TClouds (as described in the previous chapter). Many systems in production use variations of this approach to tolerate *crash faults* (e.g., [Bur06, Cal11, CGR07, Cor12, HKJR10]). Research systems have also shown that SMR can be employed with *Byzantine faults* with reasonable costs (e.g., [CL02, CKL+09, GKQV10, KBC+12, KAD+07b, VCBL09, VCB+13]).

This chapter addresses the problem of adding durability to SMR systems in general, and in BFT-SMaRt in particular. *Durability* is defined as the capability of a SMR system to survive the crash or shutdown of all its replicas, without losing any operation acknowledged to the clients. Its relevance is justified not only by the need to support maintenance operations, but also by the many examples of significant failures that occur in data centers, causing thousands of servers to crash simultaneously [Dea09, FLP+10, Mil08, Ric11].

However, the integration of durability techniques – logging, checkpointing, and state transfer – with the SMR approach can be difficult [CGR07]. First of all, these techniques can drastically decrease the performance of a service[1]. In particular, *synchronous logging* can make the system throughput as low as the number of appends that can be performed on the disk per second, typically just a few hundreds [KA08]. Although the use of SSDs can alleviate the problem, it cannot solve it completely (see §3.2.2). Additionally, *checkpointing* requires stopping the service during this operation [CL02], unless non-trivial optimizations are used at the application layer, such as copy-on-write [CGR07, CKL+09]. Moreover, recovering faulty replicas involves running a *state transfer protocol*, which can impact normal execution as correct replicas need to transmit their state.

Second, these durability techniques can complicate the programming model. In theory, SMR requires only that the service exposes an *execute()* method, called by the replication library when an operation is ready to be executed. However this leads to logs that grow forever, so in

[1]The performance results presented in the literature often exclude the impact of durability, as the authors intend to evaluate other aspects of the solutions, such as the behavior of the agreement protocol. Therefore, high throughput numbers can be observed (in req/sec) since the overheads of logging/checkpointing are not considered.

practice the interface has to support service state checkpointing. Two simple methods can be added to the interface, one to collect a snapshot of the state and another to install it during recovery. This basic setup defines a simple interface, which eases the programming of the service, and allows a complete separation between the replication management logic and the service implementation. However, this interface can become much more complex, if certain optimizations are used (see §3.2.2).

This chapter presents new techniques for implementing data durability in crash and Byzantine fault-tolerant (BFT) SMR services. These techniques are transparent with respect to both the service being replicated and the replication protocol, so they do not impact the programming model; they greatly improve the performance in comparison to standard techniques; they can be used in commodity servers with ordinary hardware configurations (no need for extra hardware, such as disks, special memories or replicas); and, they can be implemented in a modular way, as a *durability layer* placed in between the SMR library and the service.

The techniques are three: *parallel logging*, for diluting the latency of synchronous logging; *sequential checkpointing*, to avoid stopping the replicated system during checkpoints; and *collaborative state transfer*, for reducing the effect of replica recoveries on the system performance. This is the first time that the durability of fault-tolerant SMR is tackled in a *principled* way with a set of algorithms organized in an *abstraction* to be used between SMR protocols and the application.

The proposed techniques were implemented in a durability layer on the BFT-SMaRt state machine replication library [BSA], on top of which we built two services: a consistent key-value store (SCKV-Store) and a non-trivial BFT coordination service (Durable DepSpace). Our experimental evaluation shows that the proposed techniques can remove most of the performance degradation due to the addition of durability.

This chapter presents the following contributions:

1. A description of the performance problems affecting durable state machine replication, often overlooked in previous works (§3.2);
2. Three new algorithmic techniques for removing the negative effects of logging, checkpointing and faulty replica recovery from SMR, without requiring more resources, specialized hardware, or changing the service code (§3.3).
3. An analysis showing that exchanging disks by SSDs neither solves the identified problems nor improves our techniques beyond what is achieved with disks (§3.2 and §3.5);
4. The description of an implementation these techniques in BFT-SMaRt (§3.4), and an experimental evaluation under write-intensive loads, highlighting the performance limitations of previous solutions and how our techniques mitigate them (§3.5).

3.2 Durable SMR Performance Limitations

This section presents a durable SMR model, and then analyzes the effect of durability mechanisms on the performance of the system.

3.2.1 System Model and Properties

We follow the standard SMR model [Sch90]. Clients send requests to invoke operations on a service, which is implemented in a set of replicas (see Figure 3.1). Operations are executed in

the same order by all replicas, by running some form of agreement protocol. Service operations are assumed to be deterministic, so an operation that updates the state (abstracted as a *write*) produces the same new state in all replicas. The state required for processing the operations is kept in main memory, just like in most practical applications for SMR [Bur06, CGR07, HKJR10].

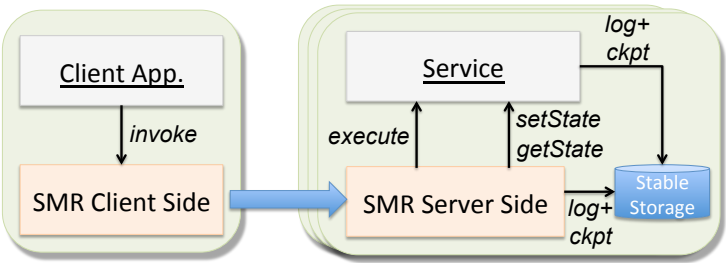


Figure 3.1: A durable state machine replication architecture.

The replication library implementing SMR has a client and a server side (layers at the bottom of the figure), which interact respectively with the client application and the service code. The library ensures standard safety and liveness properties [CL02, Lam98], such as correct clients eventually receive a response to their requests if enough synchrony exists in the system.

SMR is built under the assumption that at most f replicas fail out of a total of n replicas (we assume $n = 2f + 1$ on a crash fault-tolerant system and $n = 3f + 1$ on a BFT system). A crash of more than f replicas breaks this assumption, causing the system to stop processing requests as the necessary agreement quorums are no longer available. Furthermore, depending on which replicas were affected and on the number of crashes, some state changes may be lost. This behavior is undesirable, as clients may have already been informed about the changes in a response (i.e., the request completed) and there is the expectation that the execution of operations is persistent.

To address this limitation, the SMR system should also ensure the following property:

> *Durability:* Any request completed at a client is reflected in the service state after a recovery.

Traditional mechanisms for enforcing durability in SMR-based main memory databases are logging, checkpointing and state transfer [CGR07, GMS92]. A replica can recover from a crash by using the information saved in stable storage and the state available in other replicas. It is important to notice that a recovering replica is considered faulty *until it obtains enough data to reconstruct the state* (which typically occurs after state transfer finishes).

Logging writes to stable storage information about the progress of the agreement protocol (e.g., when certain messages arrive in Paxos-like protocols [CGR07, JRS11]) and about the operations executed on the service. Therefore, data is logged either by the replication library or the service itself, and a record describing the operation has to be stored before a reply is returned to the client.

The replication library and the service code synchronize the creation of checkpoints with the truncation of logs. The service is responsible for generating snapshots of its state (method *getState*) and for setting the state to a snapshot provided by the replication library (method *setState*). The replication library also implements a *state transfer* protocol to initiate replicas from an updated state (e.g., when recovering from a failure or if they are too late processing requests), akin to previous SMR works [CL02, CRL03, CGR07, CKL+09, RST11]. The state is fetched from the other replicas that are currently running.

3.2.2 Identifying Performance Problems

This section discusses performance problems caused by the use of logging, checkpointing and state transfer in SMR systems. We illustrate the problems with a consistent key-value store (SCKV-Store) implemented using BFT-SMART [BSA], (described in Chapter 2). In any case, the results in the chapter are mostly orthogonal to the fault model and also affect systems subject to only crash faults. We consider write-only workloads of 8-byte keys and 4kB values, in a key space of 250K keys, which creates a service state size of 1GB in 4 replicas. More details about this application and the experiments can be found in §3.4 and §3.5, respectively.

High latency of logging. As mentioned in §3.2.1, events related to the agreement protocol and operations that change the state of the service need to be logged in stable storage. Table 3.1 illustrates the effects of several logging approaches on the SCKV-Store, with a client load that keeps a high sustainable throughput:

Metric	No log	Async.	Sync. SSD	Sync. Disk
Min Lat. (ms)	1.98	2.16	2.89	19.61
Peak Thr. (ops/s)	4772	4312	1017	63

Table 3.1: Effect of logging on the SCKV-Store. Single-client minimum latency and peak throughput of 4kB-writes.

The table shows that *synchronous*[2] logging to disk can cripple the performance of such system. To address this issue, some works have suggested the use of faster non-volatile memory, such as flash memory solid state drives (SSDs) or/in NVCaches [RST11]. As the table demonstrates, there is a huge performance improvement when the log is written synchronously to SSD storage, but still only 23% of the "No log" throughput is achieved. Additionally, by employing specialized hardware, one arguably increases the costs and the management complexity of the nodes, especially in virtualized/cloud environments where such hardware may not be available in all machines.

There are works that avoid this penalty by using *asynchronous* writes to disk, allowing replicas to present a performance closer to the main memory system (e.g., Harp [LGG+91] and BFS [CL02]). The problem with this solution is that writing asynchronously does not give durability guarantees if all the replicas crash (and later recover), something that production systems need to address as correlated failures do happen [Dea09, FLP+10, Mil08, Ric11].

We would like to have a general solution that makes the performance of durable systems similar to pure memory systems, and that achieves this by *exploring the logging latency to process the requests* and by *optimizing log writes*.

Perturbations caused by checkpoints. Checkpoints are necessary to limit the log size, but their creation usually degrades the performance of the service. Figure 3.2 shows how the throughput of the SCKV-Store is affected by creating checkpoints at every 200K client requests. Taking a snapshot after processing a certain number of operations, as proposed in most works in SMR (e.g., [CL02, Lam98]), can make the system halt for a few seconds. This happens because requests are no longer processed while replicas save their state. Moreover, if the replicas are not fully synchronized, delays may also occur because the necessary agreement quorum might not be available.

[2]Synchronous writes are optimized to update only the file contents, and not the metadata, using the `rwd` mode in the Java' *RandomAccessFile* class (equivalent to using the `O_DSYNC` flag in POSIX *open*). This is important to avoid unnecessary disk head positioning.

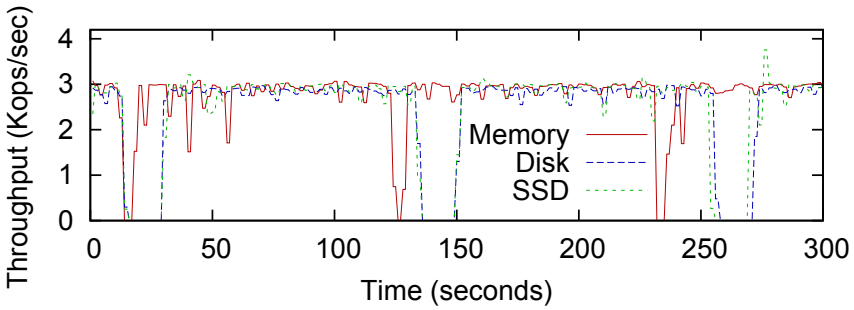


Figure 3.2: Throughput of a SCKV-Store with checkpoints in memory, disk and SSD considering a state of 1GB.

The figure indicates an equivalent performance degradation for checkpoints written in disk or SSD, meaning there is no extra benefit in using the latter (both require roughly the same amount of time to synchronously write the checkpoints). More importantly, the problem occurs even if the checkpoints are kept in memory, since the fundamental limitation is not due to storage accesses (as in logging), but to the cost to serialize a large state (1 GB).

Often, the performance decrease caused by checkpointing is not observed in the literature, either because no checkpoints were taken or because the service had a very small state (e.g., a counter with 8 bytes) [CL02, CWA+09, GKQV10, KBC+12, KAD+07b, VCBL09, VCB+13]. Most of these works were focusing on ordering requests efficiently, and therefore checkpointing could be disregarded as an orthogonal issue. Additionally, one could think that checkpoints need only to be created sporadically, and therefore, their impact is small on the overall execution. We argue that this is not true in many scenarios. For example, the SCKV-Store can process around 4700 4kB-writes per second (see §3.5), which means that the log can grow at the rate of more than 1.1 GB/min, and thus checkpoints need to be taken rather frequently to avoid outrageous log sizes. Leader-based protocols, such as those based on Paxos, have to log information about most of the exchanged messages, contributing to the log growth. Furthermore, recent SMR protocols require frequent checkpoints (every few hundred operations) to allow the service to recover efficiently from failed speculative request ordering attempts [GKQV10, KBC+12, KAD+07b].

Some systems use *copy-on-write* techniques for doing checkpointing without stoping replicas (e.g., [CKL+09]), but this approach has two limitations. First, copy-on-write may be complicated to implement at application level in non-trivial services, as the service needs to keep track of which data objects were modified by the requests. Second, even if such techniques are employed, the creation of checkpoints still consumes resources and degrades the performance of the system. For example, writing a checkpoint to disk makes logging much slower since the disk head has to move between the log and checkpoint files, with the consequent disk seek times. In practice, this limitation could be addressed in part with extra hardware, such as by using two disks per server.

Another technique to deal with the problem is *fuzzy snapshots*, used in ZooKeeper [HKJR10]. A fuzzy snapshot is essentially a checkpoint that is done without stopping the execution of operations. The downside is that some operations may be executed more than once during recovery, an issue that ZooKeeper solves by forcing all operations to be idempotent. However, making operations idempotent requires non-trivial request pre-processing before they are ordered, and increases the difficulty of decoupling the replication library from the service [HKJR10, JRS11].

We aim to have a checkpointing mechanism that *minimizes performance degradation without requiring additional hardware and, at the same time, keeping the SMR programming model*

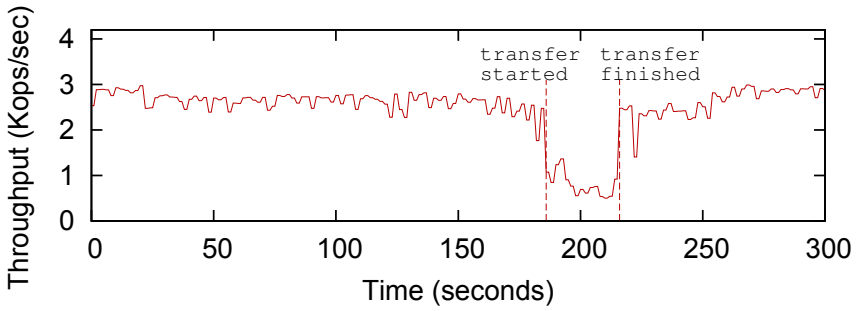


Figure 3.3: Throughput of a SCKV-Store when a failed replica recovers and asks for a state transfer.

simple.

Perturbations caused by state transfer. When a replica recovers, it needs to obtain an updated state to catch up with the other replicas. This state is usually composed of the last checkpoint plus the log up to some request defined by the recovering replica. Typically, (at least) another replica has to spend resources to send (part of) the state. If checkpoints and logs are stored in a disk, delays occur due to the transmission of the state through the network but also because of the disk accesses. Delta-checkpoint techniques based, for instance, on Merkle trees [CL02] can alleviate this problem, but cannot solve it completely since logs have always to be transferred. Moreover, implementing this kind of technique can add more complexity to the service code.

Similarly to what is observed with checkpointing, there can be the temptation to disregard the state transfer impact on performance because it is perceived to occur rarely. However, techniques such as replica rejuvenation [HKKF95] and proactive recovery [CL02, SBC+10] use state transfer to bring refreshed replicas up to date. Moreover, reconfigurations [LAB+06] and even leader change protocols (that need to be executed periodically for resilient BFT replication [CWA+09]) may require replicas to synchronize themselves [CL02, SB12]. In conclusion, state transfer protocols may be invoked much more often than when there is a crash and a subsequent recovery.

Figure 3.3 illustrates the effect of state transmission during a replica recovery in a 4 node BFT system using the PBFT's state transfer protocol [CL02]. This protocol requires just one replica to send the state (checkpoint plus log) – similarly to crash FT Paxos-based systems – while others just provide authenticated hashes for state validation (as the sender of the state may suffer a Byzantine fault). The figure shows that the system performance drops to less than 1/3 of its normal performance during the 30 seconds required to complete state transfer. While one replica is recovering, another one is slowed because it is sending the state, and thus the remaining two are unable to order and execute requests (with $f = 1$, quorums of 3 replicas are needed to order requests).

One way to avoid this performance degradation is to ignore the state transfer requests until the load is low enough to process both the state transfers and normal request ordering [HKJR10]. However, this approach tends to delay the recovery of faulty replicas and makes the system vulnerable to extended unavailability periods (if more faults occur). Another possible solution is to add extra replicas to avoid interruptions on the service during recovery [SBC+10]. This solution is undesirable as it can increase the costs of deploying the system.

We would like to have a state transfer protocol that *minimizes the performance degradation due to state transfer without delaying the recovery of faulty replicas*.

3.3 Efficient Durability for SMR

In this section we present three techniques to solve the problems identified in the previous section.

3.3.1 Parallel Logging

Parallel logging has the objective of hiding the high latency of logging. It is based on two ideas: (1) log groups of operations instead of single operations; and (2) process the operations in parallel with their storage.

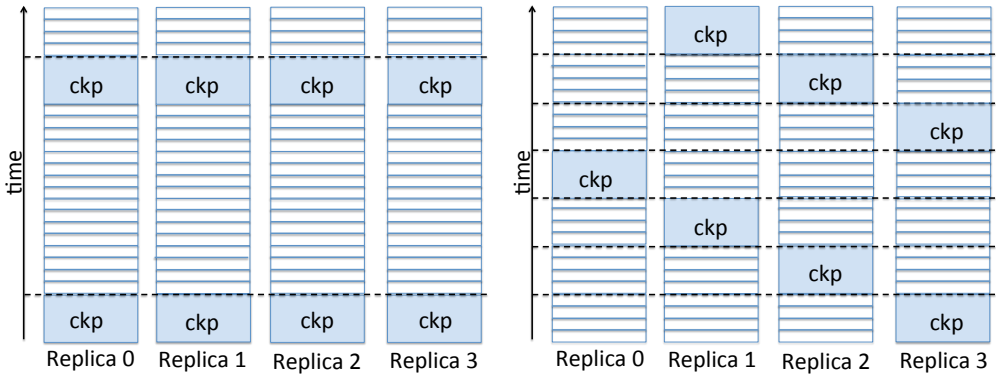
The first idea explores the fact that disks have a high bandwidth, so the latency for writing 1 or 100 log entries can be similar, but the throughput would be naturally increased by a factor of roughly 100 in the second case. This technique requires the replication library to deliver groups of service operations (accumulated during the previous batch execution) to allow the whole batch to be logged at once, whereas previous solutions normally only provide single operations, one by one. Notice that this approach is different from the batching commonly used in SMR [CL02, CWA+09, KAD+07b], where a group of operations is ordered together to amortize the costs of the agreement protocol (although many times these costs include logging a batch of requests to stable storage [Lam98]). Here the aim is to pass batches of operations from the replication library to the service, and a batch may include (batches of) requests ordered in *different agreements*.

The second idea requires that the requests of a batch are processed while the corresponding log entries are being written to the secondary storage. Notice, however, that a reply can only be sent to the client after the corresponding request is executed *and logged*, ensuring that the result seen by the client will persist even if all replicas fail and later recover. Naturally, the effectiveness of this technique depends on the relation between the time for processing a batch and the time for logging it. More specifically, the interval T_k taken by a service to process a batch of k requests is given by $T_k = max(E_k, L_k)$, where E_k and L_k represent the latency of executing and logging the batch of k operations, respectively. This equation shows that the most expensive of the two operations (execution or logging) defines the delay for processing the batch. For example, in the case of the SCKV-Store, $E_k \ll L_k$ for any k, since inserting data in a hash table with chaining (an $\mathcal{O}(1)$ operation) is much faster than logging a 4kB-write (with or without batching). This is not the case for Durable DepSpace, which takes a much higher benefit from this technique (see §3.5).

3.3.2 Sequential Checkpointing

Sequential checkpointing aims at minimizing the performance impact of taking replica's state snapshots. The key principle is to exploit the natural redundancy that exists in asynchronous distributed systems based on SMR. Since these systems make progress as long as a quorum of $n - f$ replicas is available, there are f spare replicas in fault-free executions. The intuition here is to make each replica store its state at different times, to ensure that $n - f$ replicas can continue processing client requests.

We define *global checkpointing period* P as the maximum number of (write) requests that a replica will execute before creating a new checkpoint. This parameter defines also the maximum size of a replica's log in number of requests. Although P is the same for all replicas, they checkpoint their state at different points of the execution. Moreover, all correct replicas will take at least one checkpoint within that period.



(a) Synchronized. (b) Sequential.

Figure 3.4: Checkpointing strategies (4 replicas).

An instantiation of this model is for each replica $i = 0, ..., n-1$ to take a checkpoint after processing the k-th request where $k \bmod P = i \times \lfloor \frac{P}{n} \rfloor$, e.g., for $P = 1000$, $n = 4$, replica i takes a checkpoint after processing requests $i \times 250$, $1000 + i \times 250$, $2000 + i \times 250$, and so on.

Figure 3.4 compares a synchronous (or coordinated) checkpoint with our technique. Time grows from the bottom of the figure to the top. The shorter rectangles represent the logging of an operation, whereas the taller rectangles correspond to the creation of a checkpoint. It can be observed that synchronized checkpoints occur less frequently than sequential checkpoints, but they stop the system during their execution whereas for sequential checkpointing there is always an agreement quorum of 3 replicas available for continuing processing requests.

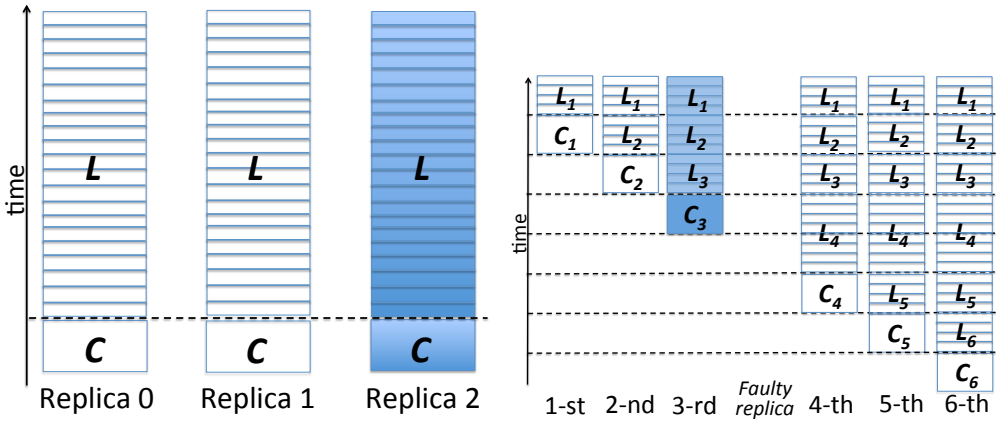
An important requirement of this scheme is to use values of P such that the chance of more than f overlapping checkpoints is negligible. Let C_{max} be the estimated maximum interval required for a replica to take a checkpoint and T_{max} the maximum throughput of the service. Two consecutive checkpoints will not overlap if:

$$\begin{aligned} C_{max} &< \frac{1}{T_{max}} \times \left\lfloor \frac{P}{n} \right\rfloor \Longrightarrow \\ P &> n \times C_{max} \times T_{max} \end{aligned} \tag{3.1}$$

Equation 3.1 defines the minimum value for P that can be used with sequential checkpoints. In our SCKV-Store example, for a state of 1GB and a 100% 4kB-write workload, we have $C_{max} \approx 15$s and $T_{max} \approx 4700$ ops/s, which means $P > 282000$. If more frequent checkpoints are required, the replicas can be organized in groups of at most f replicas to take checkpoints together.

3.3.3 Collaborative State Transfer

The state transfer protocol is used to update the state of a replica during recovery, by transmitting log records (L) and checkpoints (C) from other replicas (see Figure 3.5(a)). Typically only one of the replicas returns the full state and log, while the others may just send a hash of this data for validation (only required in the BFT case). As shown in §3.2, this approach can degrade performance during recoveries. Furthermore, it does not work with sequential checkpoints, as the received state can not be directly validated with hashes of other replicas' checkpoints (as they are different). These limitations are addressed with the *collaborative state transfer* (CST) protocol.



(a) PBFT and others ($n = 4$). (b) CST ($n = 7$).

Figure 3.5: Data transfer in different state transfer strategies.

Although the two previous techniques work both with crash-tolerant and BFT SMR, the CST protocol is substantially more complex with Byzantine faults. Consequently, we start by describing a BFT version of the protocol (which also works for crash faults) and later, at the end of the section, we explain how CST can be simplified on a crash-tolerant system[3].

We designate by *leecher* the recovering replica and by *seeders* the replicas that send (parts of) their state. CST is triggered when a replica (leecher) starts (see Figure 3.6). Its first action is to use the local log and checkpoint to determine the last logged request and its sequence number (assigned by the ordering protocol), from now on called *agreement* id. The leecher then asks for the most recent logged agreement id of the other replicas, and waits for replies until $n - f$ of them are collected (including its own id). The ids are placed in a vector in descending order, and the largest id available in $f + 1$ replicas is selected, to ensure that such agreement id was logged by at least one correct replica (steps 1-3).

In BFT-SMaRt there is no parallel execution of agreements, so if one correct replica has ordered the id-th batch, it means with certainty that agreement id was already processed by at least $f + 1$ correct replicas[4]. The other correct replicas, which might be a bit late, will also eventually process this agreement, when they receive the necessary messages.

Next, the leecher proceeds to obtain the state up to id from a seeder and the associated validation data from f other replicas. The active replicas are ordered by the freshness of the checkpoints, from the most recent to the oldest (step 4). A leecher can make this calculation based on id, as replicas take checkpoints at deterministic points, as explained in §3.3.2. We call the replica with i-th oldest checkpoint the i-th replica and the checkpoint C_i. The log of a replica is divided in segments, and each segment L_i is the portion of the log required to update the state from C_i to the more recent state C_{i-1}. Therefore, we use the following notion of equivalence: $C_{i-1} \equiv C_i + L_i$. Notice that L_1 corresponds to the log records of the requests that were executed after the most recent checkpoint C_1 (see Figure 3.5(b) for $n = 7$).

The leecher fetches the state from the $(f + 1)$-th replica (seeder), which comprises the log segments L_1, ..., L_{f+1} and checkpoint C_{f+1} (step 8). To validate this state, it also gets hashes of the log segments and checkpoints from the other f replicas with more recent checkpoints (from

[3]Even though crash fault tolerance is by far more used in production systems, our choice is justified by two factors. First, the subtleties of BFT protocols require a more extensive discussion. Second, given the lack of a stable and widely-used open-source implementation of a crash fault tolerance SMR library, we choose to develop our techniques in a BFT SMR library, so the description is in accordance to our prototype.

[4]If one employs protocols such as Paxos/PBFT, low and high watermarks may need to be considered.

1. Look at the local log to discover the last executed agreement;
2. *Fetch* the id of the last executed agreement from $n - f$ replicas (including itself) and save the identifier of these replicas;
3. id = largest agreement id that is available in $f + 1$ replicas;
4. Using id, P and n, order the replicas (including itself) with the ones with most recent checkpoints first;
5. $V \leftarrow \emptyset$; // the set containing state and log hashes
6. For $i = 1$ to f do:
   (a) *Fetch* $V_i = \langle HL_1, ..., HL_i, HC_i \rangle$ from i-th replica;
   (b) $V \leftarrow V \cup \{V_i\}$;
7. $r \leftarrow f + 1$; // replica to fetch state
8. *Fetch* $S_r = \langle L_1, ..., L_r, C_r \rangle$ from r-th replica;
9. $V \leftarrow V \cup \{\langle \mathsf{H}(S_r.L_1), ..., \mathsf{H}(S_r.L_r), \mathsf{H}(S_r.C_r) \rangle\}$;
10. Update state using $S_r.C_r$;
11. $v \leftarrow 0$; // number of validations of S_r
12. For $i = r - 1$ down to 1 do:
    (a) Replay log $S_r.L_{i+1}$;
    (b) Take checkpoint C'_i and calculate its hash HC'_i;
    (c) If $(V_i.HL_{1..i} = V_r.HL_{1..i}) \wedge (V_i.HC_i = HC'_i)$, $v++$;
13. If $v \geq f$, replay log $S_r.L_1$ and return; Else, $r++$ and go to 8;

Figure 3.6: The CST recovery protocol called by the leecher after a restart. *Fetch* commands wait for replies within a timeout and go back to step 2 if they do not complete.

the 1st until the f-th replica) (step 6a). Then, the leecher sets its state to the checkpoint and replays the log segments received from the seeder, in order to bring up to date its state (steps 10 and 12a).

The state validation is performed by comparing the hashes of the f replicas with the hashes of the log segments from the seeder and intermediate checkpoints. For each replica i, the leecher replays L_{i+1} to reach a state equivalent to the checkpoint of this replica. Then, it creates a intermediate checkpoint of its state and calculates the corresponding hash (steps 12a and 12b). The leecher finds out if the log segments sent by the seeder and the current state (after executing L_{i+1}) match the hashes provided by this replica (step 12c).

If the check succeeds for f replicas, the reached state is valid and the CST protocol can finish (step 13). If the validation fails, the leecher fetches the data from the $(f + 2)$-th replica, which includes the log segments L_1, ..., L_{f+2} and checkpoint C_{f+2} (step 13 goes back to step 8). Then, it re-executes the validation protocol, considering as extra validation information the hashes that were produced with the data from the $(f + 1)$-th replica (step 9). Notice that the validation still requires $f + 1$ matching log segments and checkpoints, but now there are $f + 2$ replicas involved, and the validation is successful even with one Byzantine replica. In the worst case, f faulty replicas participate in the protocol, which requires $2f + 1$ replicas to send some data, ensuring a correct majority and at least one valid state (log and checkpoint).

In the scenario of Figure 3.5(b), the 3rd replica (the $(f + 1)$-th replica) sends L_1, L_2, L_3 and

C_3, while the 2nd replica only transmits $HL_1 = \mathsf{H}(L_1)$, $HL_2 = \mathsf{H}(L_2)$ and $HC_2 = \mathsf{H}(C_2)$, and the 1st replica sends $HL_1 = \mathsf{H}(L_1)$ and $HC_1 = \mathsf{H}(C_1)$. The leecher next replays L_3 to get to state $C_3 + L_3$, and takes the intermediate checkpoint C'_2 and calculates the hash $HC'_2 = \mathsf{H}(C'_2)$. If HC'_2 matches HC_2 from the 2nd replica, and the hashes of log segments L_2 and L_1 from the 3rd replica are equal to HL_2 and HL_1 from the 2nd replica, then the first validation is successful. Next, a similar procedure is applied to replay L_2 and the validation data from the 1st replica. Now, the leecher only needs to replay L_1 to reach the state corresponding to the execution of request id.

While the state transfer protocol is running, replicas continue to create new checkpoints and logs since the recovery does not stop the processing of new requests. Therefore, they are required to keep old log segments and checkpoints to improve their chances to support the recovery of a slow leecher. However, to bound the required storage space, these old files are eventually removed, and the leecher might not be able to collect enough data to complete recovery. When this happens, it restarts the algorithm using a more recent request id (a similar solution exists in all other state state transfer protocols that we are aware of, e.g., [CL02, CGR07]).

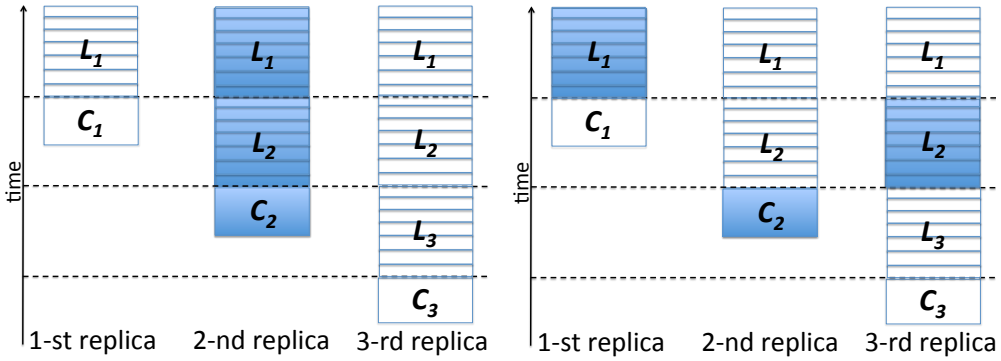
The leecher observes the execution of the other replicas while running CST, and stores all received messages concerning agreements more recent than id in an out-of-context buffer. At the end of CST, it uses this buffer to catch up with the other replicas, allowing it to be reintegrated in the state machine.

Correctness. We present here a brief correctness argument of the CST protocol. Assume that b is the actual number of faulty (Byzantine) replicas (lower or equal to f) and r the number of recovering replicas.

In terms of safety, the first thing to observe is that CST returns if and only if the state is validated by at least $f + 1$ replicas. This implies that the state reached by the leecher at the end of the procedure is valid according to at least one correct replica. To ensure that this state is recent, the largest agreement id that is returned by $f + 1$ replicas is used.

Regarding liveness, there are two cases to consider. If $b+r \leq f$, there are still $n-f$ correct replicas running and therefore the system could have made progress while the r replicas were crashed. A replica is able to recover as long as checkpoints and logs can be collected from the other replicas. Blocking is prevented because CST restarts if any of the *Fetch* commands fails or takes too much time. Consequently, the protocol is live if correct replicas keep the logs and checkpoints for a sufficiently long interval. This is a common assumption for state transfer protocols. If $b + r > f$, then there may not be enough replicas for the system to continue processing. In this case the recovering replica(s) will continuously try to fetch the most up to date agreement id from $n - f$ replicas (possibly including other recovering replicas) until such quorum exists. Notice that a total system crash is a special case of this scenario.

Optimizing CST for $f = 1$. When $f = 1$ (and thus $n = 4$), a single recovering replica can degrade the performance of the system because one of $n - f$ replicas will be transferring the checkpoint and logs, delaying the execution of the agreements (as illustrated in Figure 3.7(a)). To avoid this problem, we spread the data transfer between the active replicas through the following optimization in an *initial recovery round*: the 2nd replica ($f + 1 = 2$) sends C_2 plus $\langle HL_1, HL_2 \rangle$ (instead of the checkpoint plus full log), while the 1st replica sends L_1 and HC_1 (instead of only hashes) and the 3rd replica sends L_2 (instead of not participating). If the validation of the received state fails, then the normal CST protocol is executed. This optimization is represented in Figure 3.7(b), and in §3.5 we show the benefits of this strategy.



(a) General CST. (b) Optimized CST.

Figure 3.7: General and optimized CST with $f = 1$.

Simplifications for crash faults. When the SMR only needs to tolerate crash faults, a much simpler version of CST can be employed. The basic idea is to execute steps 1-4 of CST and then fetch and use the checkpoint and log from the 1st (most up to date) replica, since no validation needs to be performed. If $f = 1$, a analogous optimization can be used to spread the burden of data transfer among the two replicas: the 1st replica sends the checkpoint while the 2nd replica sends the log segment.

3.4 Implementation: Dura-SMaRt

In order to validate our techniques, we extended the open-source BFT-SMART replication library [BSA] with a durability layer, placed between the request ordering and the service. We named the resulting system *Dura-SMaRt*, and used it to implement two applications: a consistent key-value store and a coordination service.

Adding durability to BFT-SMART. BFT-SMaRt originally offered an API for invoking and executing state machine operations, and some callback operations to fetch and set the service state. The implemented protocols are described in [SB12] and follow the basic ideas introduced in PBFT and Aardvark [CL02, CWA+09]. BFT-SMaRt is capable of ordering more than 100K 0-byte msg/s (the 0/0 microbenchmark used to evaluate BFT protocols [GKQV10, KAD+07b]) in our environment. However, this throughput drops to 20K and 5K msgs/s for 1kB and 4kB message sizes, respectively (the workloads we use – see §3.5).

We modified BFT-SMaRt to accommodate an intermediate *Durability layer* implementing our techniques at the server-side, as described in Figure 3.8, together with the following modifications on BFT-SMaRt. First, we added a new server side operation to deliver batches of requests instead of one by one. This operation supplies ordered but not delivered requests spanning one or more agreements, so they can be logged in a single write by the *Keeper* thread. Second, we implemented the parallel checkpoints and collaborative state transfer in the *Dura-Coordinator* component, removing the old checkpoint and state transfer logic from BFT-SMaRt and defining an extensible API for implementing different state transfer strategies. Finally, we created a dedicated thread and socket to be used for state transfer in order to decrease its interference on request processing.

SCKV-store. The first system implemented with Dura-SMaRt was a *simple and consistent key-value store* (SCKV-Store) that supports the storage and retrieval of key-value pairs, alike to

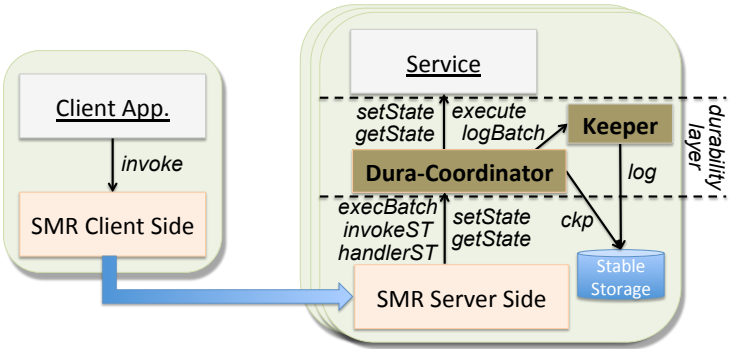


Figure 3.8: The Dura-SMaRt architecture.

other services described in the literature, e.g., [CST+10, ORS+11]. The implementation of the SCKV-Store was greatly simplified, since consistency and availability come directly from SMR and durability is achieved with our new layer.

Durable DepSpace (DDS). The second use case is a durable extension of the DepSpace coordination service [BACF08], which originally stored all data only in memory. The system, named Durable DepSpace (DDS), provides a tuple space interface in which tuples (variable-size sequences of typed fields) can be inserted, retrieved and removed. There are two important characteristics of DDS that differentiate it from similar services such as Chubby [Bur06] and ZooKeeper [HKJR10]: it does not follow a hierarchical data model, since tuple spaces are, by definition, unstructured; and it tolerates Byzantine faults, instead of only crash faults. The addition of durability to DepSpace basically required the replacement of its original replication layer by Dura-SMaRt.

3.5 Evaluation

This section evaluates the effectiveness of our techniques for implementing durable SMR services. In particular, we devised experiments to answer the following questions: (1) What is the cost of adding durability to SMR services? (2) How much does *parallel logging* improve the efficiency of durable SMR with synchronous disk and SSD writes? (3) Can *sequential checkpoints* remove the costs of taking checkpoints in durable SMR? (4) How does *collaborative state transfer* affect replica recoveries for different values of f? Question 1 was addressed in §3.2, so we focus on questions 2-4.

Case studies and workloads. As already mentioned, we consider two SMR-based services implemented using Dura-SMaRt: the SCKV-Store and the DDS coordination service. Although in practice, these systems tend to serve mixed or read-intensive workloads [CST+10, HKJR10], we focus on write operations because they stress both the ordering protocol and the durable storage (disk or SSD). Reads, on the other hand, can be served from memory, without running the ordering protocol. Therefore, we consider a 100%-write workload, which has to be processed by an agreement, execution and logging. For the SCKV-Store, we use YCSB [CST+10] with a new workload composed of 100% of replaces of 4kB-values, making our results comparable to other recent SMR-based storage systems [BBH+11, RST11, WAD12]. For DDS, we consider the insertion of random tuples with four fields containing strings, with a total size of 1kB, creating a workload with a pattern equivalent to the ZooKeeper evaluation [HKJR10, JRS11].

Experimental environment. All experiments, including the ones in §3.2, were executed in a cluster of 14 machines interconnected by a gigabit ethernet. Each machine has two quad-core 2.27 GHz Intel Xeon E5520, 32 GB of RAM memory, a 146 GB 15000 RPM SCSI disk and a 120 GB SATA Flash SSD. We ran the IOzone benchmark[5] on our disk and SSD to understand their performance under the kind of workload we are interested: rewrite (append) for records of 1MB and 4MB (the maximum size of the request batch to be logged in DDS and SCKV-Store, respectively). The results are:

Record length	Disk	SSD
1MB	96.1 MB/s	128.3 MB/s
4MB	135.6 MB/s	130.7 MB/s

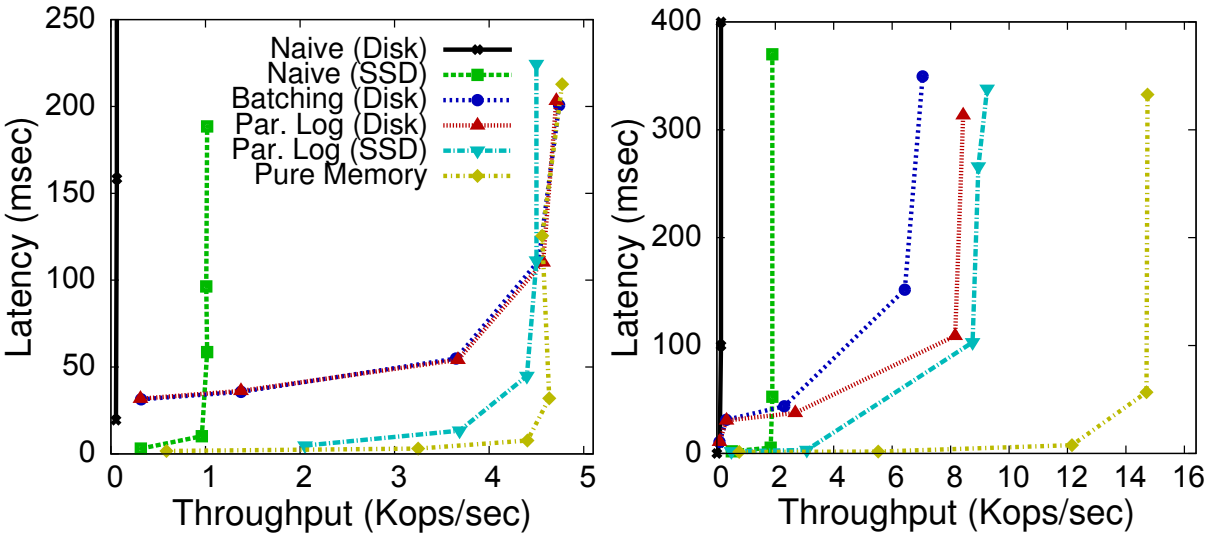
Parallel logging. Figure 3.9(a) displays latency-throughput curves for the SCKV-Store considering several durability variants. The figure shows that naive (synchronous) disk and SSD logging achieve a throughput of 63 and 1017 ops/s, respectively, while a pure memory version with no durability reaches a throughput of around 4772 ops/s.

Parallel logging involves two ideas, the storage of batches of operations in a single write and the execution of operations in parallel with the secondary storage accesses. The use of batch delivery alone allowed for a throughput of 4739 ops/s with disks (a $75\times$ improvement over naive disk logging). This roughly represents what would be achieved in Paxos [KA08, Lam98], ZooKeeper [HKJR10] or UpRight [CKL+09], with requests being logged during the agreement protocol. Interestingly, the addition of a separated thread to write the batch of operations, does not improve the throughput of this system. This occurs because a local put on SCKV-Store replica is very efficient, with almost no effect on the throughput.

The use of parallel logging with SSDs improves the latency of the system by 30-50ms when compared with disks until a load of 4400 ops/s. After this point, parallel logging with SSDs achieves a peak throughput of 4500 ops/s, 5% less than parallel logging with disk (4710 ops/s), with the same observed latency. This is consistent with the IOzone results. Overall, parallel logging with disk achieves 98% of the throughput of the pure memory solution, being the replication layer the main bottleneck of the system. Moreover, the use of SSDs neither solves the problem that parallel logging addresses, nor improves the performance of our technique, being thus not effective in eliminating the log bottleneck of durable SMR.

Figure 3.9(b) presents the results of a similar experiment, but now considering DDS with the same durability variants as in SCKV-Store. The figure shows that a version of DDS with naive logging in disk (resp. SSD) achieves a throughput of 143 ops/s (resp. 1900 ops/s), while a pure memory system (DepSpace), reaches 14739 ops/s. The use of batch delivery improves the performance of disk logging to 7153 ops/s (a $50\times$ improvement). However, differently from what happens with SCKV-Store, the use of parallel logging in disk further improves the system throughput to 8430 ops/s, an improvement of 18% when compared with batching alone. This difference is due to the fact that inserting a tuple requires traversing many layers [BACF08] and the update of an hierarchical index, which takes a non-negligible time (0.04 ms), and impacts the performance of the system if done sequentially with logging. The difference would be even bigger if the SMR service requires more processing. Finally, the use of SSDs with parallel logging in DDS was more effective than with the SCKV-Store, increasing the peak throughput of the system to 9250 ops/s (an improvement of 10% when compared with disks). Again, this is consistent with our IOzone results: we use 1kB requests here, so the batches are smaller than in SCKV-Store, and SSDs are more efficient with smaller writes.

[5] http://www.iozone.org.



(a) SCKV-Store. (b) Durable DepSpace.

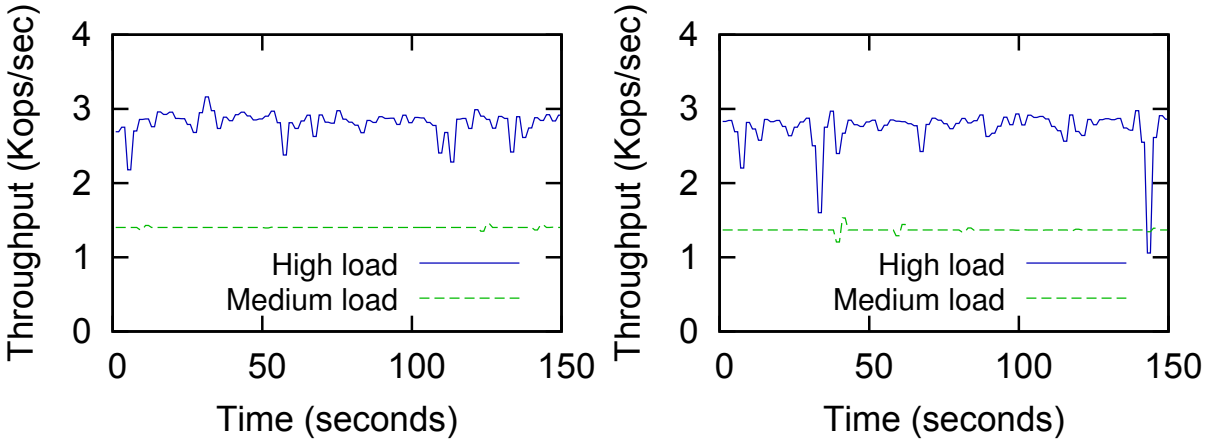
Figure 3.9: Latency-throughput curves for several variants of the SCKV-Store and DDS considering 100%-write workloads of 4kB and 1kB, respectively. Disk and SSD logging are always done synchronously. The legend in (a) is valid also for (b).

Notice that DDS could not achieve a throughput near to pure memory. This happens because, as discussed in §3.3.1, the throughput of parallel logging will be closer to a pure memory system if the time required to process a batch of requests is akin to the time to log this batch. In the experiments, we observed that the workload makes BFT-SMaRt deliver batches of approximately 750 requests on average. The local execution of such batch takes around 30 ms, and the logging of this batch on disk entails 70 ms. This implies a maximum throughput of 10.750 ops/s, which is close to the obtained values. With this workload, the execution time matches the log time (around 500 ms) for batches of 30K operations. These batches require the replication library to reach a throughput of 60K 1kB msgs/s, three times more than what BFT-SMaRt achieves for this message size.

Sequential Checkpointing. Figure 3.10 illustrates the effect of executing sequential checkpoints in disks with SCKV-Store[6] during a 3-minute execution period.

When compared with the results of Figure 3.2 for synchronized checkpoints, one can observe that the unavailability periods no longer occur, as the 4 replicas take checkpoints separately. This is valid both when there is a high and medium load on the service and with disks and SSDs (not show). However, if the system is under stress (high load), it is possible to notice a periodic small decrease on the throughput happening with both 500MB and 1GB states (Figures 3.10(a) and 3.10(b)). This behavior is justified because at every $\lfloor \frac{P}{n} \rfloor$ requests one of the replicas takes a checkpoint. When this occurs, the replica stops executing the agreements, which causes it to become a bit late (once it resumes processing) when compared with the other replicas. While the replica is still catching up, another replica initiates the checkpoint, and therefore, a few agreements get delayed as the quorum is not immediately available. Notice

[6]Although we do not show checkpoint and state transfer results for DDS due to space constraints, the use of our techniques bring the same advantage as on SCKV-Store. The only noticeable difference is due to the fact that DDS local tuple insertions are more costly than SCKV-Store local puts, which makes the variance on the throughput of sequential checkpoints even more noticeable (especially when the leader is taking its checkpoint). However, as in SCKV-Store, this effect is directly proportional to the load imposed to the system.



(a) 500MB state. (b) 1GB state.

Figure 3.10: SCKV-Store throughput with sequential checkpoints with different write-only loads and state size.

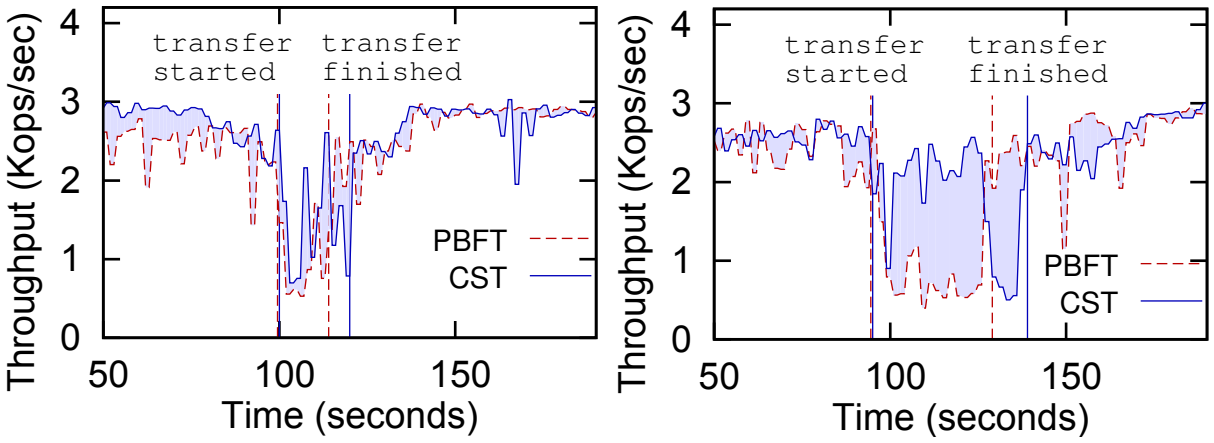
that this effect does not exist if the system has less load or if there is sufficient time between sequential checkpoints to allow replicas to catch up ("Medium load" line in Figure 3.10).

Collaborative State Transfer. This section evaluates the benefits of CST when compared to a PBFT-like state transfer in the SCKV-Store with disks, with 4 and 7 replicas, considering two state sizes. In all experiments a single replica recovery is triggered when the log size is approximately twice the state size, to simulate the condition of Figure 3.7(b).

Figure 3.11 displays the observed throughput of some executions of a system with $n = 4$, running PBFT and the CST algorithm optimized for $f = 1$, for states of 500MB and 1GB, respectively. A PBFT-like state transfer takes 30 (resp. 16) seconds to deliver the whole 1 GB (resp. 500MB) of state with a sole replica transmitter. In this period, the system processes 741 (resp. 984) write ops/sec on average. CST optimized for $f = 1$ divides the state transfer by three replicas, where one sends the state and the other two up to half the log each. Overall, this operation takes 42 (resp. 20) seconds for a state of 1GB (resp. 500MB), 28% (resp. 20%) more than with the PBFT-like solution for the same state size. However, during this period the system processes 1809 (resp. 1426) ops/sec on average. Overall, the SCKV-Store with a state of 1GB achieves only 24% (or 32% for 500MB-state) of its normal throughput with a PBFT-like state transfer, while the use of CST raises this number to 60% (or 47% for 500MB-state).

Two observations can be made about this experiment. First, the benefit of CST might not be as good as expected for small states (47% of the normal throughput for a 500MB-state) due to the fact that when fetching state from different replicas we need to wait for the slowest one, which always brings some degradation in terms of time to fetch the state (20% more time). Second, when the state is bigger (1GB), the benefits of dividing the load among several replicas make state transfer much less damaging to the overall system throughput (60% of the normal throughput), even considering the extra time required for fetching the state ($+28\%$).

We did an analogous experiment for $n = 7$ (not shown due to space constraints) and observed that, as expected, the state transfer no longer causes a degradation on the system throughput (both for CST and PBFT) since state is fetched from a single replica, which is available since $n = 7$ and there is only one faulty replica (see Figure 3.5). We repeated the experiment for $n = 7$ with the state of 1GB being fetched from the leader, and we noticed a 65% degradation on the throughput. A comparable effect occurs if the state is obtained from the leader in CST. As a cautionary note, we would like to remark that when using spare replicas



(a) 500MB and $n = 4$.

(b) 1GB and $n = 4$.

Figure 3.11: Effect of a replica recovery on SCKV-Store throughput using CST with $f = 1$ and different state sizes.

for "cheap" faulty recovery, it is important to avoid fetching the state from the leader replica (as in [Bur06, CGR07, HKJR10, RST11]) because this replica dictates the overall system performance.

3.6 Related Work

Over the years, there has been a reasonable amount of work about stable state management in main memory databases (see [GMS92] for an early survey). In particular, parallel logging shares some ideas with classical techniques such as group commit and pre-committed transactions [DKO+84] and the creation of checkpoints in background has also been suggested [Lam91]. Our techniques were however developed with the SMR model in mind, and therefore, they leverage the specific characteristics of these systems (e.g., log groups of requests while they are executed, and schedule checkpoints preserving the agreement quorums).

Durability management is a key aspect of practical crash-FT SMR-like systems [BBH+11, CGR07, HKJR10, JRS11, RST11, WAD12]. In particular, making the system use the disk efficiently usually requires several hacks and tricks (e.g., non-transparent copy-on-write, request throttling) on an otherwise small and simple protocol and service specification [CGR07]. These systems usually resort to dedicated disks for logging, employ mostly synchronized checkpoints and fetch the state from a leader [CGR07, HKJR10, RST11]. A few systems also delay state transfer during load-intensive periods to avoid a noticeable service degradation [HKJR10, WAD12]. All these approaches either hurt the SMR elegant programming model or lead to the problems described in §3.2.2. For instance, recent consistent storage systems such as Windows Azure Storage [Cal11] and Spanner [Cor12] use Paxos together with several extensions for ensuring durability. We believe works like ours can improve the modularity of future systems requiring durable SMR techniques.

BFT SMR systems use logging, checkpoints, and state transfer, but the associated performance penalties often do not appear in the papers because the state is very small (e.g., a counter) or the checkpoint period is too large (e.g., [CL02, CWA+09, GKQV10, KBC+12, KAD+07b, VCBL09, VCB+13]). A notable exception is UpRight [CKL+09], which implements durable state machine replication, albeit without focusing on the efficiency of logging, checkpoints and state transfer. In any case, if one wants to sustain a high-throughput (as reported in the papers)

for non-trivial states, the use of our techniques is fundamental. Moreover, any implementation of proactive recovery [CL02, SBC+10] requires an efficient state transfer.

PBFT [CL02] was one of the few works that explicitly dealt with the problem of optimizing checkpoints and state transfer. The proposed mechanism was based on copy-on-write and delta-checkpoints to ensure that only pages modified since the previous checkpoint are stored. This mechanism is complementary to our techniques, as we could use it together with the sequential checkpoints and also to fetch checkpoint pages in parallel from different replicas to improve the state transfer. However, the use of copy-on-write may require the service definition to follow certain abstractions [CRL03, CKL+09], which can increase the complexity of the programming model. Additionally, this mechanism, which is referred in many subsequent works (e.g., [GKQV10, KAD+07b]), only alleviates but does not solve the problems discussed in §3.2.2.

A few works have described solutions for fetching different portions of a database state from several "donors" for fast replica recovery or database cluster reconfiguration (e.g., [KBB01]). The same kind of techniques were employed for fast replica recovery in group communication systems [KZHR07] and, more recently, in main-memory-based storage [ORS+11]. There are three differences between these works and ours. First, these systems try to improve the recovery time of faulty replicas, while CST main objective is to minimize the effect of replica recovery on the system performance. Second, we are concerned with the interplay between logging and checkpoints, which is fundamental in SMR, while these works are more concerned with state snapshots. Finally, our work has a broader scope in the sense that it includes a set of complementary techniques for Byzantine and crash faults in SMR systems, while previous works address only crash faults.

3.7 Conclusion

This chapter discusses several performance problems caused by the use of logging, checkpoints and state transfer on SMR systems, and proposes a set of techniques to mitigate them. The techniques – parallel logging, sequential checkpoints and collaborative state transfer – are purely algorithmic, and require no additional support (e.g., hardware) to be implemented in commodity servers. Moreover, they preserve the simple state machine programming model, and thus can be integrated in any crash or Byzantine fault-tolerant library without impact on the supported services.

The techniques were implemented in a durability layer for BFT-SMaRt, which was used to develop two representative services: a KV-store and a coordination service. Our results show that these services can reach up to 98% of the throughput of pure memory systems, remove most of the negative effects of checkpoints and substantially decrease the throughput degradation during state transfer. We also show that the identified performance problems can not be solved by exchanging disks by SSDs, highlighting the need for techniques such as the ones presented here.

Chapter 4

Evaluating BFT-SMaRt over a WAN

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

4.1 Introduction

As already discussed in previous chapters, the State Machine Replication technique [Lam78, Sch90] enables an arbitrary number of client to issue requests into a set of replicas. These replicas implement a stateful service that updates its state after receiving those requests. The goal of this technique is to enforce strong consistency within the service, by making it completely and accurately replicated at each replica. The key to make the state evolve with such consistency is to execute a distributed protocol that forces each operation sent by clients to be delivered at each replica in the exact same order. When clients get the service's reply, their requests are already reflected in the service's state.

This chapter reports preliminary results regarding the performance of a Byzantine fault-tolerant (BFT) state machine replication (SMR) protocol executing over a wide area network (WAN). Besides evaluating the protocol's performance, we also implemented and evaluated some optimizations from the literature which aim to render executions of SMR protocols more stable and efficient across WAN environments. The experiments presented in this chapter were made using BFT-SMaRt [BSA] (described in Chapter 2), a library that implements a generic BFT SMR protocol similar to the well known PBFT [CL02].

The main motivation behind these preliminary experiments is twofold. First, we wanted to evaluate the behavior of a generic BFT SMR protocol when deployed on a large scale environment, which represents a cloud-of-clouds scenario. Secondly, there is significant research in optimizing SMR for large scale environments (e.g., [ADD+10, MJM08, VCBL10]) which propose several optimizations to SMR aimed at improving performance in a WAN. We decided to perform our own evaluation of some of the optimizations introduced in those works. We did this by implementing them in a generic BFT SMR protocol (used by BFT-SMaRt).

Both LAN and WAN settings assume the same system model; some hosts are assumed to be malicious, and the network connecting each host does not offer any time guarantees. This is in accordance to the assumptions made by BFT-SMaRt protocol [SB12] (also TClouds D2.2.2). The practical difference is that, in a WAN setting, network delay is much higher and variable, and message loss is much more frequent than it is in a LAN.

Our preliminary results suggest that a generic BFT SMR protocols can display stable performance across WANs. However, out of the three optimizations evaluated, only one seemed to significantly improve the protocol's performance; the remaining two did not introduce any observable improvement. Further experiments will be required in order to reach conclusive results.

The rest of this chapter is organized as follows. We discuss some related work in §5.6. §4.3 presents the hypotheses we aim to investigate. §4.4 describes the methodology used to conducted the experiments. We report our results for each hypothesis in §4.5, §4.6, §4.7 and §4.8. Finally, we discuss future work in §4.9 and present our conclusions in §4.10.

4.2 Related Work

In this section we give a brief overview of related work regarding Byzantine fault tolerance, state machine replication, wide area replication and wide area measurements.

Byzantine fault tolerance Byzantine fault tolerance (BFT) is a sub-field of fault tolerance research within distributed systems. In classical fault tolerance, processes are assumed to fail only by stopping to execute. On the other hand, in the *Byzantine faults model*, processes of a distributed system are allowed to fail in a arbitrary way, i.e., a fault is characterized as any deviation from the specified algorithm/protocol imposed on a process. Thus, Byzantine fault tolerance is the body of techniques that aims at devising protocols, algorithms and services that are able to cope with such arbitrary behavior of the processes that comprise the system [LSP82]. In Byzantine fault tolerance it is common practice to make assumptions as weak as possible, not only from the processes that comprise the system, but also from the network that connects them. In BFT, it is typically adopted either the partially synchronous or asynchronous system model [HT94].

Castro and Liskov showed that executing SMR under Byzantine faults is feasible, by presenting the PBFT protocol [CL02]. PBFT is capable of withstanding up to f Byzantine replicas out of at least $3f + 1$. The main difference between PBFT and previous proposals for BFT were that PBFT avoided expensive cryptographic operations such as digital signatures by using MAC vectors instead. PBFT spawned a *renaissance* in BFT research, and is considered the baseline for all BFT SMR protocols published afterwards. Moreover, the idea of porting SMR protocols into WANs also found new momentum.

Wide Area Replication Mencius [MJM08] is a SMR protocol derived from Paxos [Lam98, Les01] which is optimized to execute in WANs. It can survive up to f crashed replicas out of at least $2f + 1$. According to the paper, its rotating coordinator mechanism can significantly reduce clients' latency in WANs. Replicas take turns as the leader and propose client requests in their turns. Clients send requests to the replicas in their sites, which are submitted for ordering when the replicas becomes the leader.

Veronese et. al. introduced *Efficient Byzantine Agreement for Wide-Area Networks* (EBAWA) [VCBL10], a BFT SMR protocol optimized for WANs. Since it uses the trusted/trustworthy USIG service introduced in [VCB+13], it requires only $2f + 1$ replicas to tolerate f Byzantine faults. Similarly to Mencius, it uses a rotating leader scheme to prevent a faulty leader from degrading system performance.

Measurements The aforementioned replication protocols rely on some form of quorum systems to be capable of guaranteeing their safety properties. Given a set of hosts, a *quorum system* is a collection of sets of hosts (called *quorums*) such that any two quorums intersect in at least one common host [Gif79, GMB85]. Since quorum systems are building blocks used to implement a variety of services (e.g., consensus [Cac09], mutual exclusion [AA91], distributed access control [NW98]), there is interest in predicting their availability and performance in WANs.

The traditional approaches for evaluating quorum systems used to be either by analysis and/or by emulation. Amir et. al. proposed in [AW96] an empirical approach, which consisted of gathering uptime data from a real system consisting of multiple hosts. These hosts were scattered across two distinct sites which communicated with each other over the internet. The aforementioned data was obtained using a group communication framework responsible for generating uptime logs for each host in the experiment. These logs were then integrated into one single global log, which represented the availability of all hosts within some time period. According to the authors, the results obtained suggest that machine crashes are correlated, network partitions are frequent, and a system crash is a rare, yet possible event. They also confirm that, if network partitions are frequent, dynamic quorum systems (e.g., [Her87]) yield better availability than static quorums.

Bakr et. al. also employed empirical measurements in [BK02]. Whereas Amir et. al. investigated the *availability* of a quorum system, Bakr et. al. investigate the *latency* of distributed algorithms over the internet. Both works used real hosts to obtain their measurements, but the methodology was different: instead of using a communication group framework to construct logs, Bakr et. al. implemented their own daemons that would periodically initiate the algorithms, and keep track of the running time for each iteration. Furthermore, the hosts were geographically distributed across more than two sites. Upon evaluating some of these algorithms, the authors observed the message loss rate over the internet was not negligible. Moreover, algorithms with high message complexity display higher loss rate. This is evident in protocols that employ *all-to-all* communication, like PBFT and BFT-SMaRt.

However, the experiments performed in [BK02] did not assume quorum systems, i.e., each interaction of a communication round was only considered finished once every host received and replied all messages. In a quorum system, each hosts only waits for a majority of hosts to receive/reply to all messages. The authors conducted further research in [BK08] considering quorum systems and how the number of hosts probed by a client impacts latency and availability. Furthermore, two quorum types were studied: (1) majority quorum systems, where the quorums are sets that include a majority of the hosts, and (2) crumbling walls quorum systems[1] [PW97], which uses smaller quorums with varying sizes. The authors argue that their results suggest that majority quorums do not perform well with small probe sets. Moreover, the authors also claim that increasing the size of the probe by as few as a single host can reduce latency by a considerable margin. They also argue that their results show that crumbling walls can display better latency than majority quorums, and also comparable availability. On the other hand, this study only investigated availability of quorums as a function of the probing performed by clients; the behavior of complex distributed protocols which execute over quorum systems was not explored.

Finally, it is worth mentioning the experiments presented in [AW96, BK02, BK08] were conducted from 1996 to 2008. Given the current advances in network infrastructures, the same experiments may yield different results if they were executed in 2013.

4.3 Hypotheses

In the experiments reported in this chapter, we evaluated some protocol optimizations known in the literature, which are implemented by the SMR protocols described in §5.6-c, and one

[1] In a crumbling wall, hosts are logically arranged in rows of varying widths, and a quorum is the union of one full row and a representative from every row below the full row.

optimization related to quorum systems proposed in [Gif79, Pâr86]. More precisely, we want to test the following hypotheses:

1. The leader location influences the observed latency of the protocol (§4.5);
2. A bigger quorum size can reduce the observed latency (§4.6);
3. Read-only and tentative executions significantly reduces the observed latency (§4.7).

Finally, we evaluated the stability of BFT-SMaRt's protocol within a WAN, i.e., how much the latency observed by BFT-SMaRt clients vary across a long execution. More specifically, the following hypothesis was tested in §4.8: *A generic BFT SMR protocol is stable and predictable enough in a WAN environment and can be used to implement services capable of exhibiting satisfactory performance and usability*.

4.4 Methodology

The following experiments were conducted over the course of approximately 40 days on Planetlab, a distributed testbed scattered throughout the world, dedicated to computer networking and distributed systems research. A total of eight host were selected for these experiments. These hosts are listed in Table 4.1, and were divided into groups A and B, each one executing a single instance of a simple benchmarking application implemented over BFT-SMaRt.

Group A is comprised of a set of 6 replicas, whereas group B is comprised of 4. Furthermore, replica 1 is represented in 2 hosts. The reasons for this are the following: (1) one extra host was necessary for the experiment described in §4.6, and (2) the original host for replica 1 (Italy, Parma) became unavailable during the course of the experiments. Hence, we needed to deploy a new host to take its place (Braga, Portugal).

No additional hosts were used to run the clients. Each host ran two processes simultaneously: one executed a BFT-SMaRt replica, and the other ran multiple threads which implemented BFT-SMaRt clients. Each client sent a request to the replicas every 2 seconds. A distinguished client at each host was programmed to write its observed latency into a log file. Each client issued a 4 kB request, and received a 4 kB reply.

4.5 Leader Location

The goal of this first experiment is to observe how much the leader's proximity to an aggregate of clients can improve their observed latency. This is motivated by the fact that Mencius [MJM08] and EBAWA [VCBL10] both use a rotating leader scheme to improve client's latency: if the leader is close to the clients, their messages arrive sooner and thus are ordered faster.

This experiment ran on Group B with an aggregate of 10 clients. The experiment was repeated 4 times, each one placing the aggregate in a distinct host. Each iteration took approximately one day to run. Only one client was launched on the rest of the hosts that did not had the aggregate.

Table 4.2 presents the average latencies observed by client in each distinct location. Each row depicts the observed latency for the client location, whereas each column depicts the location of the aggregate. In the case of Gliwice - where clients run in the same machine as the

Group A

Replica	Location	Hostname
0 (leader)	Birmingham, England	planetlab1.aston.ac.uk
1	Italy, Parma Portugal, Braga	planet1.unipr.it planetlab-um00.di.uminho.pt
2	Germany, Darmstadt	host2.planetlab.informatik.tu-darmstadt.de
3	Norway, Oslo	planetlab1.ifi.uio.no
4	Belgium, Namur	orval.infonet.fundp.ac.be

Group B

Replica	Location	Hostname
0 (leader)	Poland, Gliwice	plab4.ple.silweb.pl
1	Spain, Madrid	utet.ii.uam.es
2	France, Paris	ple3.ipv6.lip6.fr
3	Switzerland, Basel	planetlab-1.cs.unibas.ch

Table 4.1: Hosts used in experiments

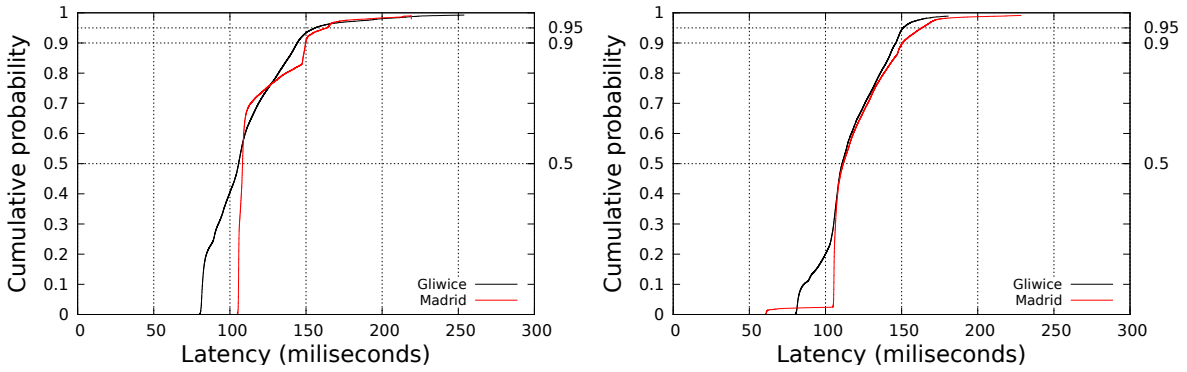
leader - the average latency was lowest when the aggregate was placed in the same host as the leader. Moreover, for the rest of the locations, latency was highest when they also hosted the aggregate (only exception being Basel). This indicates that moving an aggregate of clients close to the leader may improve latency. However, if the leader is in a distinct site, client latency tends to increase at the site that hosts the aggregate.

Client \ Aggregate	Gliwice	Madrid	Paris	Basel
Gliwice	112 ± 54.07	117 ± 211.02	113 ± 34.44	118 ± 39.61
Madrid	120 ± 44.78	122 ± 158.38	122 ± 150.28	90 ± 42.12
Paris	120 ± 68.64	123 ± 230.96	125 ± 90.71	94 ± 56.56
Basel	119 ± 182.55	122 ± 151.83	119 ± 33.50	96 ± 173.31

Table 4.2: Average latency and standard deviation observed in Group B's distinguished clients, with the leader in Gliwice. All values are given in milliseconds.

Figure 4.1 presents the cumulative distribution for the latencies observed by distinguished clients in Gliwice and Madrid. Figure 4.1(a) illustrate the latency when the aggregate is close to the leader (both located in Gliwice), whereas 4.1(b) illustrate the latency observed once the leader and aggregate are detached (leader located in Gliwice, aggregate in Madrid).

Despite placing the aggregate close to the leader, the above results suggest that doing so just barely improves client latency (e.g, moving the aggregate from Madrid to Gliwice results in a average improvement of less than 3%). There are two possible explanations for such small benefit: a) BFT-SMART's protocol needs to execute two *all-to-all* communication steps, which combined take much more time to finish than the time it takes for the clients to contact all replicas (regardless of their geographic location). Mencius and EBAWA are able to avoid such communication complexity because, unlike BFT-SMART, they do not assume a full BFT model (Mencius assumes crash faults and EBAWA uses a trusted component); b) The size of the message sent on the one-to-all communication step initialized by the leader is much larger than any of the all-to-all communication steps sent afterwards (since they only contain a cryptographic



(a) Aggregate in Gliwice.

(b) Aggregate in Madrid.

Figure 4.1: CDF of latencies with aggregates in different locations and leader in Gliwice.

hash of the requests being ordered). The results shown in §4.7 suggests this is the correct explanation. Given these results, it does not seem to be advantageous to periodically change the leader's location.

4.6 Quorum Size

The purpose of this experiment was to observe how client latency is affected by the quorum size demanded by the protocol. This is motivated by the works of Gifford [Gif79] and Pâris [Pâr86], which use voting schemes with additional hosts to improve availability. While Gifford makes all hosts hold a copy of the state with distinct voting weights, Pâris makes a distinction between hosts that hold a copy of the state and hosts that do not hold such copy, but still participate in the voting process (thus acting solely as witnesses).

This experiment ran on Group A, with replica 1 hosted in Braga. It was repeated twice: one using 4 hosts and other using 5. BFT-SMART was modified to use only 3 replicas out of the whole set in each iteration (thus, in each execution waiting for 3/4 and 3/5 replicas respectively). Each experiment executed for approximately 24 hours. Five clients were launched at each host, creating a total of 20 clients in the experiment.

Client Location	Quorum size	
	3/4	3/5
Birmingham	551 ± 1440.44	812 ± 1319.025
Braga	258 ± 132.83	171 ± 38.3
Darmstadt	266 ± 149.89	200 ± 291.88
Oslo	260 ± 131.027	203 ± 203.39

Table 4.3: Average latency and standard deviation observed in Group A. All values are given in milliseconds.

The results shown in Table 4.3 display the average latency and standard deviation observed in the distinguished clients in both iterations. After running the experiment with one more replica in the group, both average latency and standard deviation decreased in all distinguished clients (with the exception of Birmingham). Since BFT-SMART still waits for the same number of replies in each communication step, the slowest replica within that set is replaced by the

additional one, thus decreasing latency. Birmingham did not display this behavior because the host was visibly unstable throughout both iterations.

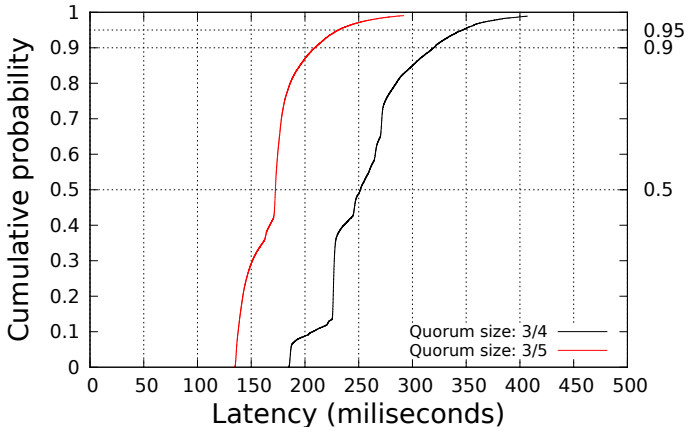


Figure 4.2: Cumulative distribution of latencies observed in Braga (group A).

The difference between both iterations is better illustrated in Figure 4.2, which displays the cumulative distribution function of the latency observed in Braga's distinguished client. When using 3/4 quorums, the 95th percentile of all latencies is approximately 350 milliseconds, whereas when using 3/5 the same percentile is approximately 240 milliseconds. This observation further indicates that using additional replicas can improve the latency observed by clients.

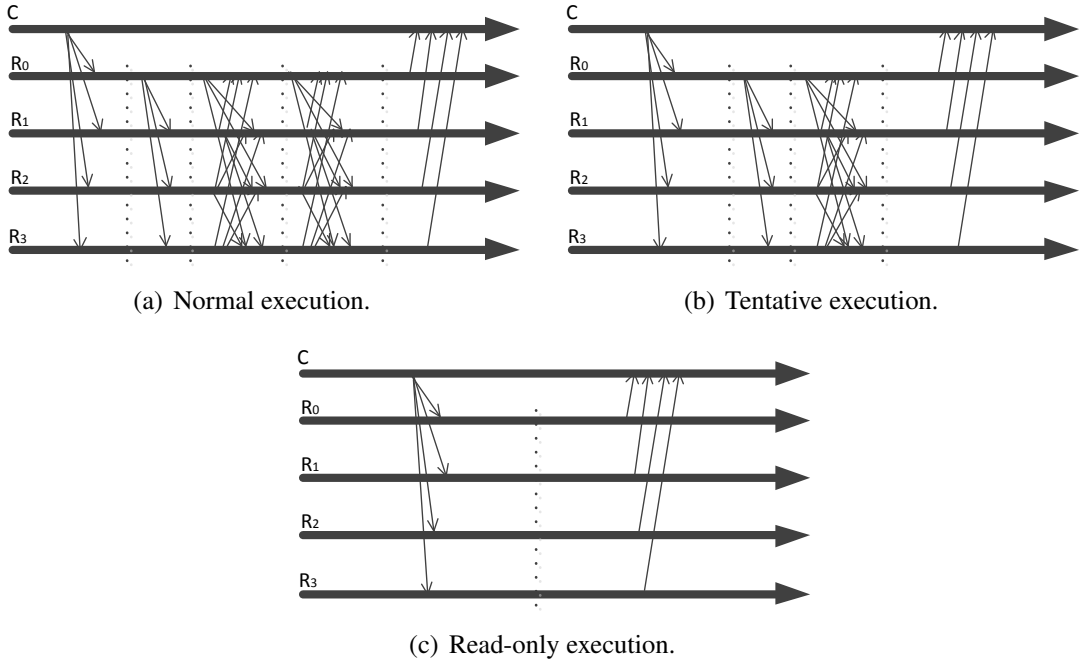
4.7 Communication Steps

The purpose of the following experiments is to observe how client latency is affected by the amount of communication steps performed by the BFT SMR protocol. More precisely, we wanted to observe how efficient *read-only* and *tentative* executions are in a WAN. These two optimizations are proposed in PBFT [CL02] to reduce latency. The message pattern for each of these optimizations is illustrated in Figure 4.3.

Figure 4.3(a) depicts the normal execution of our generic SMR protocol, which is comprised of 5 communications steps (as in BFT-SMaRt). Figure 4.3(b) displays the message pattern for tentative executions. This optimization reduces the number of communication steps from 4 to 5 by bypassing one of the all-to-all communication steps of normal execution. This optimization comes at the cost of potentially needing to perform a rollback on the application state. Figure 4.3(c) shows the message pattern for read-only executions. This optimization enables the clients to fetch a response from the service in only 2 communication steps (the client's request and the replicas' replies). However, this optimization can only be used to read the state from the service, and never to modify it.

The experiment here reported ran on Group B. To perform the tentative executions, BFT-SMaRt was modified to skip one of the all-to-all communication steps specified by its protocol [SB12]. Read-only executions were already implemented in BFT-SMaRt. Each iteration was executed for approximately 12 hours. Five clients were launched at each host, thus creating a total of 20 clients in the experiment.

Table 4.4 shows the average latency and standard deviation observed by the distinguished clients in each iteration. Figure 4.4 depicts the cumulative distribution for latencies observed



(a) Normal execution.

(b) Tentative execution.

(c) Read-only execution.

Figure 4.3: Message patterns evaluated.

Client Location	Execution Type		
	Read Only	Tentative	Normal
Gliwice	59 ± 11.14	100 ± 35.79	112 ± 29.33
Madrid	31 ± 6.86	112 ± 37.09	122 ± 183.54
Paris	25 ± 154.32	110 ± 39.38	118 ± 43.61
Basel	33 ± 224.7	110 ± 35.78	117 ± 30.315

Table 4.4: Average latency and standard deviation observed in Group B. All values are given in milliseconds.

by Madrid's distinguished clients for each type of execution. Both Table 4.4 and Figure 4.4 show that read-only execution significantly exhibits the lowest latency, finishing each execution faster than any of the other iterations (in Madrid's case, as less as 25.4% of the latency of normal execution).

Tentative execution also manages to reach lower latency values than normal execution does. However, even though this optimization omits an entire all-to-all communication step, the difference is not significant (less than 8% improvement on average). This may be explained by the fact that the size of the message sent on the one-to-all communication step initialized by the leader is much larger than any of the all-to-all communication steps performed afterwards (which only contains a cryptographic hash of the requests being ordered). In any case, the results for this particular experiment are inconclusive, and will be further investigated in future work.

4.8 BFT-SMaRt Stability

The goal of this experiment was to observe how stable BFT-SMaRt's protocol is once deployed in a WAN, and to find if there is a time interval within which it is highly likely to finish ordering requests. This was done by observing the latencies experienced by distinguished

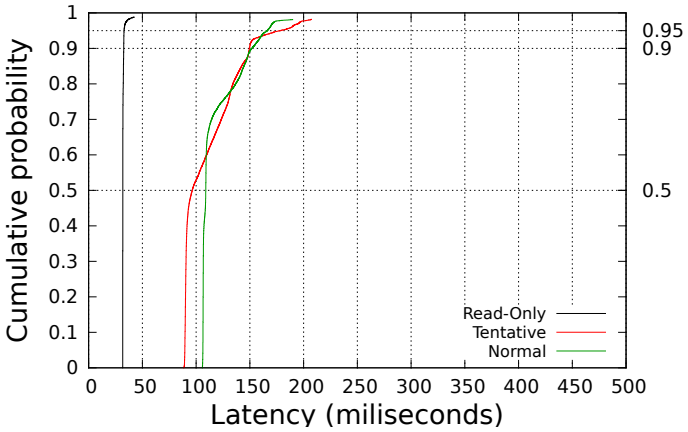


Figure 4.4: Cumulative distribution of latencies observed in Madrid (group B).

clients. None of the optimizations evaluated in Sections 4.5-4.7 were used in this experiment.

This experiment was executed within group A over a period of approximately 600 hours. In this experiment, replica 1 was hosted in Parma, Italy. Five clients were launched at each host, thus making a total of 20 clients in the experiment.

Client Location	Invocations	Average Latency	Median	90th	95th
Birmingham	717782	195 ± 144.61	172	287	436
Parma	918293	210 ± 237.81	183	280	462
Darmstadt	444054	203 ± 207.54	172	305	459
Oslo	708811	220 ± 166.16	192	293	444

Table 4.5: Average latency, standard deviation, Median, 90th and 95th percentile observed in Group A. Values are given in milliseconds.

Table 4.5 shows the results for the average latency, standard deviation, Median, 90th and 95th percentile calculated at each distinguished client. It also discriminates the number of invocations performed by each one of them. Figure 4.5 plots the cumulative distribution for those latencies. During this 600-hour experiment (comprising more than 2.7M measurements), the average latency ranged from 195 to 220 milliseconds across all sites. Even though these averages fall in an acceptable range, their associated standard deviations are high, ranging from 144.51 to 237.81. This demonstrates that latency was quite variable during this experiment.

On the other hand, about 95% of the observed latencies fall under 462 milliseconds at all locations. This means that latency, albeit variable, rarely exceeds 500 milliseconds. Even though it is approximately 5 times higher than the ideal latency of 100 miliseconds identified in [Nie93, Mil68, CRM91], clients received their response under much less than one second in the 95th percentile (which according to the same studies, is still sufficient for user satisfaction). This suggests that BFT SMR protocols can be stable enough to be used across WANs and are able to reply to clients within a predictable time interval.

4.9 Discussion & Future Work

Out of the three optimizations evaluated, only the quorum size was shown to be effective in improving the clients's latency. Our 600-hour experiment indicated that clients experience latency

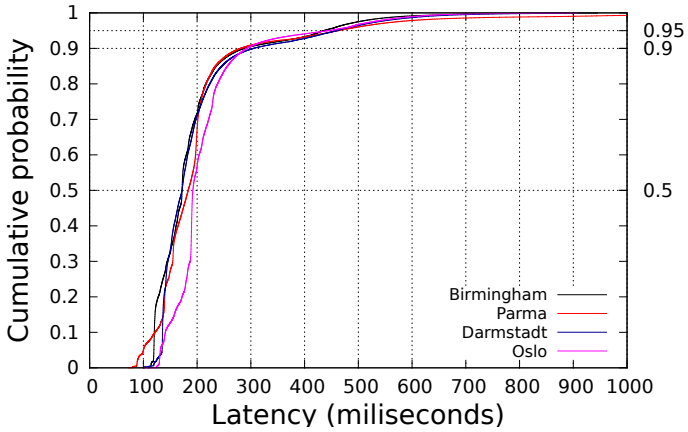


Figure 4.5: Cumulative distribution of latencies of group A over the course of two weeks.

that remains within user satisfaction limits. Both optimizations tested in §4.5 and §4.7 seem to be ineffective due to the message size sent by clients. There is one more explanation for such lack of performance gain: a 4 kB payload is likely to cause packet fragmentation, which implies more probability of packet loss. Since we use TCP connections, the packets are retransmitted, but the remaining ones may get hold in TCP buffers. Given that we are working within a WAN, this is a possible explanation for these results. In the future we plan to repeat these experiments using 1 kB of payload in client messages.

4.10 Conclusion

In this chapter we have reported preliminary results from experiments conducted in PlanetLab executing a state machine replication protocol capable of withstanding Byzantine faults. These experiments were meant to find how would a standard BFT SMR protocol benefit from optimizations taken from the literature, and to learn how it would perform over a WAN environment. Albeit further work is necessary, these results indicate that out of the three optimizations evaluated, using smaller quorums is the one that yields best performance enhancement. Using a rotating leader scheme does not seem to bring any benefit, and it is inconclusive whether or not removing communication steps improves performance. Finally, we showed that a standard BFT SMR protocol can display sufficiently predictable and acceptable latency in a WAN, which shows evidence that the cloud-of-clouds paradigm can be used in practice.

Chapter 5

FITCH: Supporting Adaptive Replicated Services in the Cloud

Chapter Authors:
Vinícius Cogo, André Nogueira, João Sousa and Alysson Bessani (FFCUL).

5.1 Introduction

Dynamic resource provisioning is one of the most significant advantages of cloud computing. Elasticity is the property of adapting the amount and capacity of resources, which makes it possible to optimize performance and minimize costs under highly variable demands. While there is widespread support for dynamic resource provisioning in cloud management infrastructures, adequate support is generally missing for service replication, in particular for systems based on state machine replication [CL02, Sch90].

Replication infrastructures typically use a static configuration of replicas. While this is adequate for replicas hosted on dedicated servers, the deployment of replicas on cloud providers creates novel opportunities. Replicated services can benefit from dynamic adaptation as replica instances can be added or removed dynamically, the size and capacity of the resources available to replicas can be changed, and replica instances may be migrated or replaced by different instances. These operations can lead to benefits such as increased performance, increased security, reduced costs, and improved legal conformity. Managing cloud resources for a replication group creates additional requirements for resource management and demands a coherent coordination of adaptations with the replication mechanism.

In this chapter we present FITCH (*Fault- and Intrusion-Tolerant Cloud computing Hardpan*), a novel infrastructure to support dynamic adaptation of replicated services in cloud environments. FITCH aggregates several components found in cloud infrastructures and some new ones in a *hybrid architecture* [Ver02] that supports *fundamental operations* required for adapting replicated services considering dependability, performance and cost requirements. A key characteristic of this architecture is that it can be easily deployed in current data centres [HB09] and cloud platforms.

We validate FITCH with two representative replicated services: a crash-tolerant web service providing static content and a Byzantine fault-tolerant (BFT) key-value store based on state machine replication (using BFT-SMaRt). When deployed in our FITCH infrastructure, we were able to easily extend both services for improved dependability through proactive recovery [CL02, SNV05] and rejuvenation [HKKF95] with minimal overhead. Moreover, we also show that both services can reconfigure and adapt to varying workloads, through horizontal (adding/removing replicas) and vertical (upgrading/downgrading replicas) scalability.

FITCH fills a gap between works that propose either (a) protocols for reconfiguring repli-

cated services [LMZ10b, LAB+06] or (b) techniques for deciding when and how much to adapt a service based on its observed workload and predefined SLA [BAP07, DPC10, Gar04]. Our work defines a system architecture that can receive adaptation commands provided by (b) and leverages cloud computing flexibility in providing resources by orchestrating the required reconfiguration actions. FITCH provides a basic interface for adding, removing and replacing replicas, coordinating all low level actions mentioned providing end-to-end service adaptability. The main contributions of this chapter are:

- A systematic analysis of the motivations and technical solutions for dynamic adaptation of replicated services deployed the cloud (§5.2);
- The FITCH architecture, which provides generic support for dynamic adaptation of replicated services running in cloud environments (§5.3 and §5.4);
- An experimental demonstration that efficient dynamic adaptation of replicated services can be easily achieved with FITCH for two representative services and the implementation of proactive recovery, and horizontal and vertical scalability (§5.5).

5.2 Adapting Cloud Services

We review several technical solutions regarding dynamic service adaptation and correlate them with motivations to adapt found in production systems, which we want to satisfy with FITCH.

Horizontal scalability is the ability of **increasing or reducing the number of computing instances** responsible for providing a service. An increase – scale-out – is an action to deal with peaks of client requests and to increase the number of faults the system can tolerate. A decrease – scale-in – can save resources and money. Vertical scalability is achieved through upgrade and downgrade procedures that respectively **increase and reduce the size or capacity of resources allocated** to service instances (e.g., Amazon EC2 offers predefined categories for VMs – small, medium, large, and extra large – that differ in CPU and memory resources [Ama06]). Upgrades – scale-up – can improve service capacity while maintaining the number of replicas. Downgrades – scale-down – can release over-provisioned allocated resources and save money.

Moving replicas to different cloud providers can result in performance improvements due to different resource configurations, or financial gains due to different prices and policies on billing services, and is beneficial to prevent vendor lock-in [BCQ+11]. **Moving service instances close to clients** can bring relevant performance benefits. More specifically, logical proximity to the clients can reduce service access latency. **Moving replicas logically away from attackers** can increase the network latency experienced by the attacker, reducing the impact of its attacks on the number of requests processed (this can be especially efficient for denial-of-service attacks).

Replacing faulty replicas (crashed, buggy or compromised) is a reactive process following fault detections, which replaces faulty instances by new, correct ones [SBC+10]. It decreases the costs for the service owner, since he still has to pay for faulty replicas, removing also a potential performance degradation caused by them, and restores the service fault tolerance. **Software replacement** is an operation where software such as operating systems and web servers are replaced in all service instances at run-time. Different implementations might differ on performance aspects, licensing costs or security mechanisms. **Software update** is the process of replacing software in all replicas by up-to-date versions. Vulnerable software, for instance, must be replaced as soon as patches are available. New software versions may also increase

performance by introducing optimized algorithms. In systems running for long periods, long running effects can cause performance degradation. Software rejuvenation can be employed to **avoid such ageing problems** [HKKF95].

5.3 The FITCH Architecture

In this section, we present the architecture of the FITCH infrastructure for replicated services adaptation.

5.3.1 System and Threat Models

Our system model considers a usual cloud-based Infrastructure-as-a-Service (IaaS) with a large pool of physical machines hosting user-created virtual machines (VMs) and some trusted infrastructure for controlling the cloud resources [HB09]. We assume a *hybrid distributed system* model [Ver02], in which different components of the system follow different fault and synchrony models. Each machine that hosts VMs may fail arbitrarily, but it is equipped with a trusted subsystem that can fail only by crashing (i.e., cannot be intruded or corrupted). Some machines used to control the infrastructure are trusted and can only fail by crashing. In addition, the network interconnecting the system is split into two isolated networks, here referred to as *data plane* and *control plane*.

User VMs employ the *data plane* to communicate internally and externally with the internet. All components connected to this network are untrusted, i.e., can be subject to Byzantine faults [CL02] (except the service gateway, see §5.3.3). We assume this network and the user VMs follow a *partially synchronous* system model [DLS88]. The *control plane* connects all trusted components in the system. Moreover, we assume that this network and the trusted components follow a *synchronous* system model with bounded computations and communications. Notice that although clouds do not employ real-time software and hardware, in practice the over provision of the control network associated with the use of dedicated machines, together with over provisioned VMs running with high priorities, makes the control plane a "de facto" synchronous system (assuming sufficiently large time bounds) [RK07, SBC+10].

5.3.2 Service Model

FITCH supports replicated services running on the untrusted domain (as user VMs) that may follow different replication strategies. In this chapter we focus on two extremes of a large spectrum of replication solutions.

The first extreme is represented by *stateless services* in which the replicas rely on a shared storage component (e.g., a database) to store their state. Notice that these services do have a state, but they do not need to maintain it coherently. Service replicas can process requests without contacting other replicas. A server can fetch the state from the shared storage after recovering from a failure, and resume processing. Classical examples of stateless replicated services are web server clusters in which requests are distributed following a load balancing policy, and the content served is fetched from a shared distributed file system.

The other extreme is represented by consistent *stateful services* in which the replicas coordinate request execution following the *state machine replication* model [CL02, Sch90]. In this model, an arbitrary number of client processes issue commands to a set of replica processes.

These replicas implement a stateful service that changes its state after processing client commands, and sends replies to the issuing clients. All replicas have to execute the same sequence of commands, which requires the use of an agreement protocol to establish a total order before request execution. The Paxos-based coordination and storage systems used by Google [CGR07] are examples of services that follow this model.

One can fit most popular replication models between these two extreme choices, such as the ones providing eventual consistency, used, for example, in Amazon's Dynamo [DHJ+07]. Moreover, the strategies we consider can be used together to create a dependable multi-tier architecture. Its clients connect to a stateless tier (e.g., web servers) that executes operations and, when persistent state access is required, they access a stateful tier (e.g., database or file system) to read or modify the state.

5.3.3 Architecture

The FITCH architecture, as shown in Fig. 5.1, comprises two subsystems, one for controlling the infrastructure and one for client service provisioning. All components are connected either with the control plane or the data plane. In the following we describe the main architectural components deployed on these domains.

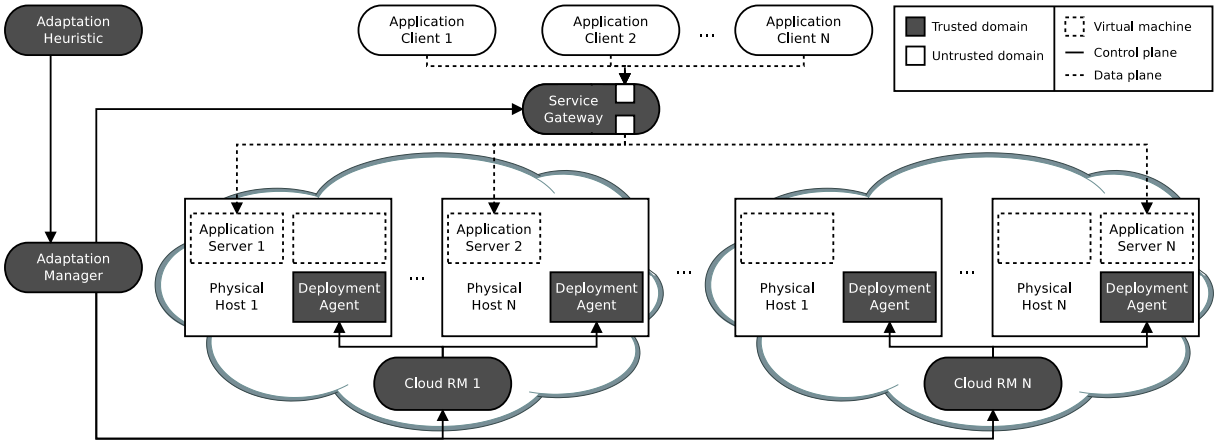


Figure 5.1: The FITCH architecture.

The **trusted domain** contains the components used to control the infrastructure. The core of our architecture is the *adaptation manager*, a component responsible to perform the requested dynamic adaptations in the cloud-hosted replicated services. This component is capable of inserting, removing and replacing replicas of a service running over FITCH. It provides public interfaces to be used by the *adaptation heuristic*. Such heuristic defines when and plans how adaptations must be performed, and is determined by human administrators, reactive decision engines, security information event managers and other systems that may demand some dynamic adaptation on services.

The *service gateway* maintains the group membership of each service and supports pluggable functionalities such as proxy and load balancing. It works like a lookup service, where new clients and instances request the most up-to-date group of replicas of a given service through the data plane. The adaptation manager informs the service gateway about each modification in the service membership through the control plane. The service gateway thus is a special component connected to both networks. This is a reasonable assumption as all state

updates happen via the trusted control plane, whereas clients have read-only access. Moreover, having a trusted point of contact for fetching membership data is a common assumption in dynamic distributed systems [LMZ10b].

Cloud resource managers (RM) provide resource allocation capabilities based on requests from the adaptation manager in order to deploy or destroy service replicas. The adaptation manager provides information about the amount of resources to be allocated for a specific service instance and the VM image that contains all required software for the new replica. The cloud RM chooses the best combination of resources for that instance based on requested properties. This component belongs to the trusted domain, and the control plane carries out all communication involving the cloud RM.

The *deployment agent* is a small trusted component, located inside cloud physical hosts, which is responsible to guarantee the deployment of service instances. It belongs to the trusted domain and receives deployment requests from cloud RM through the control plane. The existence of this component follows the paradigm of hybrid nodes [Ver02], previously used in several other works (e.g., [DPSP+11, RK07, SNV05, SBC+10]).

The **untrusted domain** contains the components that provide the adaptive replicated service. *Application servers* are virtual machines used to provide the replicated services deployed on the cloud. Each of these user-created VMs contains all software needed by its service replica. Servers receive, process and answer client requests directly or through the service gateway depending on the configuration employed.

Cloud physical hosts are physical machines that support a virtualization environment for server consolidation. These components host VMs containing the replicas of services running over FITCH. They contain a hypervisor that controls the local resources and provides strong isolation between hosted VMs. A cloud RM orchestrates such environment through the control plane.

Application clients are processes that perform requests to service instances in order to interact with the replicated service. They connect to the service gateway component to discover the list of available replicas for a given service (and later access them directly) or send requests directly to the service gateway, which will forward them to the respective service instances. The application clients can be located anywhere in the internet. They belong to the untrusted domain and communicate with the application servers and gateway through the data plane.

5.3.4 Service Adaptation

The FITCH architecture described in the previous section supports adaptation on deployed replicated services as long as the adaptation manager, the gateway and some cloud resource managers are available. This means that during unavailability of these components (due to an unexpected crash, for instance), the replicated service can still be available, but adaptation operations are not. Since these services are deployed on a trusted and synchronous subsystem, it is relatively simple to implement fault tolerance for them, even transparently using VM-based technology [CLM+08].

The FITCH infrastructure supports three basic adaptation operations for a replicated service: *add*, *remove* and *replace* a replica. All adaptation solutions defined in §5.2 can be implemented by using these three operations. When the request to adapt some service arrives at the adaptation manager, it triggers the following sequence of operations (we assume a single replica is changed, for simplicity):

1. If adding or replacing:

1.1. The adaptation manager contacts the cloud RM responsible for the physical host that matches the requested criteria for the new replica. The cloud RM informs the deployment agent on the chosen physical host asking it to create a new VM with a given image (containing the new replica software). When the VM is launched, and the replica process is started, the deployment agent informs the cloud RM, which informs the adaptation manager that a new replica is ready to be added to the service.

1.2. The adaptation manager informs the gateway that it needs to *reconfigure* the replicated service to add the newly created replica for the service. The gateway invokes a reconfiguration command on the service to add the new replica.[1] When the reconfiguration completes, the gateway updates the current group membership information of the service.

2. If removing or replacing:

2.1. The adaptation manager informs the gateway that it needs to *reconfigure* the replicated service to remove a service replica. The gateway invokes a reconfiguration command on the service to remove the old replica.[1] When the reconfiguration completes, the gateway updates the group membership of the service.

2.2. The adaptation manager contacts the cloud RM responsible for the replica being removed or replaced. The cloud RM asks the deployment agent of the physical host in which the replica is running to destroy the corresponding VM. At the end of this operation, the cloud RM is informed and then it passes this information to the adaptation manager.

Notice that, for a replica replacement, the membership needs to be updated twice, first adding the new replica and then removing the old one. We intentionally use this two-step approach for all replacements, because it simplifies the architecture and is necessary to guarantee services' liveness and fault tolerance.

5.4 Implementation

We implemented the FITCH architecture and two representative services to validate it. The services are a crash fault-tolerant (CFT) web service and a consistent BFT key-value store.

FITCH. Cloud resource managers are the components responsible for deploying and destroying service VMs, following requests from the adaptation manager. Our prototype uses OpenNebula, an open source system for managing virtual storage, network and processing resources in cloud environments. We decided to use Xen as a virtualization environment that controls the physical resources of the service replicas. The deployment agent is implemented as a set of scripts that runs on a separated privileged VM, which has no interface with the untrusted domain.

A service gateway maintains information about the service group. In the stateless service, the service gateway is a load balancer based on Linux Virtual Server [Zha00], which redirects client requests to application servers. In our stateful service implementation, an Apache Tomcat web server provides a basic service lookup.

Our adaptation manager is a Java application that processes adaptation requests and communicates with cloud RMs and the service gateway to address those requests. The communication between the adaptation manager and cloud resource managers is done through OpenNebula API for Java. Additionally, the communication between the adaptation manager and the service gateway is done through secure sockets (SSL).

In our implementation, each application server is a virtual machine running Linux. All software needed to run the service is present in the VM image deployed at each service instance. Different VLANs, in a Gigabit Ethernet switch, isolate data and control planes.

[1] The specific command depends on the replication technique and middleware being used by the service. We assume that the replication middleware implements a mechanism that ensures a consistent reconfiguration (e.g., [LMZ10b, LAB+06]).

Stateless service. In the replicated stateless web service, each client request is processed independently, unrelated to any other requests previously sent to any replica. It is composed of some number of replicas, which have exactly the same service implementation, and are orchestrated by a load balancer in the service gateway, which forwards clients requests to be processed by one of the replicas.

Stateful service. In the key-value store based on BFT state machine replication [CL02], each request is processed in parallel by all service replicas and a correct answer is obtained by voting on replicas replies. To obtain such replication, we developed our key-value store over a Byzantine state machine replication library called BFT-SMaRt [BSA] (see Chapter 2). Such key-value store is basically the SCKV-store described in Chapter 3, but here configured to store operation logs and state checkpoints in main memory. For the purpose of this report, it is enough to know that BFT-SMaRt employs a leader-based total order protocol similar to PBFT [CL02] and that it implements a reconfiguration protocol following the ideas presented by Lamport *et al.* [LMZ10b].

Adaptations. We implemented three adaptation solutions using the FITCH infrastructure. Both services employ *proactive recovery* [CL02, SNV05] by periodically replacing a replica by a new and correct instance. This approach allows a fault-tolerant system to tolerate an arbitrary number of faults in the entire service life span. The window of vulnerability in which faults in more than f replicas can disrupt a service is reduced to the time it takes all hosts to finish a recovery round. We use *horizontal scalability* (adding and removing replicas) for the stateless service, and we analyse *vertical scalability* (upgrading and downgrading the replicas) of the stateful service.

5.5 Experimental Evaluation

We evaluate our implementation in order to quantify the benefits and the impact caused by employing dynamic adaptation in replicated services running in cloud environments. We first present the experimental environment and tools, followed by experiments to measure the impact of proactive recovery, and horizontal and vertical scalability.

5.5.1 Experimental Environment and Tools

Our experimental environment uses 17 physical machines that host all FITCH components (see Table 5.1). This cloud environment provides three types of virtual machines – *small* (1 CPU, 2GB RAM), *medium* (2 CPU, 4GB RAM) or *large* (4 CPU, 8GB RAM). Our experiments use two benchmarks. The stateless service is evaluated using the WS-Test [Sun04] web services microbenchmark. We executed the echoList application within this benchmark, which sends and receives linked lists of twenty 1KB elements. The stateful service is evaluated using YCSB [CST+10], a benchmark for cloud-serving data stores. We implemented a wrapper to translate YCSB calls to requests in our BFT key-value store and used three workloads: a read-heavy workload (95% of GET and 5% of PUT [CST+10]), a pure-read (100% GET) and a pure-write workload (100% PUT). We used OpenNebula version 2.0.1, Xen 3.2-1, Apache Tomcat 6.0.35 and VM images with Linux Ubuntu Intrepid and kernel version 2.6.27-7-server for x86_64 architectures.

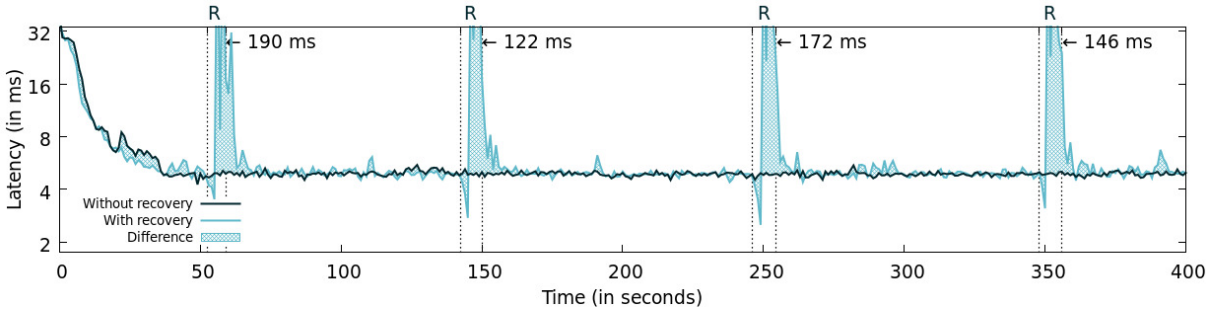


Figure 5.2: Impact on a CFT stateless service. Note that the y-axis is in logarithmic scale.

5.5.2 Proactive Recovery

Our first experiment consists in replacing the entire set of service replicas as if implementing software rejuvenation or proactive/reactive recovery [CL02, HKKF95, RK07, SBC+10]. The former is important to avoid software ageing problems, whereas the latter enforces service's fault tolerance properties (for instance, the ability to tolerate f faults) [SNV05].

The idea of this experiment is to recover all n replicas from the service group, one-by-one, as early as possible, without disrupting the service availability (i.e., maintaining $n - f$ active replicas). Each recovery consists of creating and adding a new replica to the group and removing an old replica from it. We perform n recoveries per experiment, where n is the number of service replicas, which depends on the type and number of faults to be tolerated, and on the protocol used to replace all replicas. The time needed to completely recover a replica can also vary for each reconfiguration protocol.

Impact on a CFT stateless web service. Our first application in this experiment is a stateless web service that is initially composed of 4 large (L) replicas (which can tolerate 3 crash faults). The resulting latencies (in ms) are presented in Fig. 5.2, with and without the recovering operations in an experiment that took 400 s to finish. In this graph, each replica recovery is marked with a "R" symbol and has two lines representing the beginning and the end of replacements, respectively. The average of service latency without recovery was 5.60 ms, whereas with recovery was 8.96 ms. This means that the overall difference in the execution with and without recoveries is equivalent to 60% (represented in the filled area of Fig. 5.2). However, such difference is mostly caused during replacements, which only happens during 7.6% of the execution time.

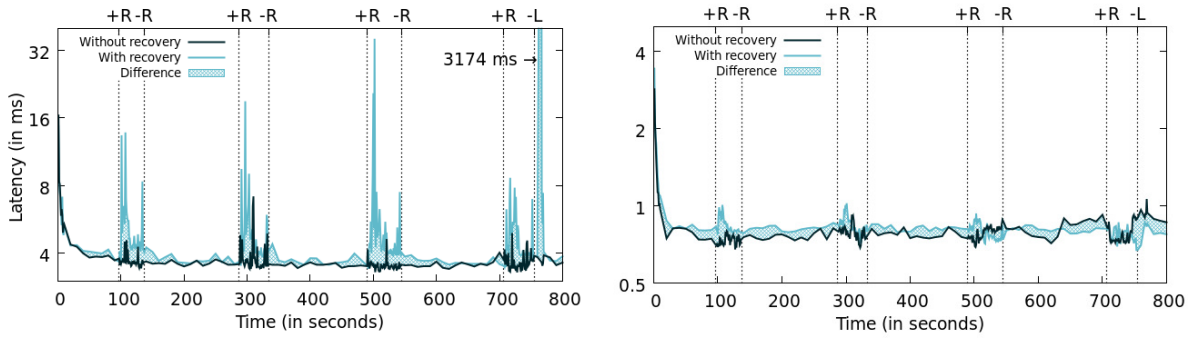
We draw attention to three aspects of the graph. First, the service has an initial warm-up phase that is independent of the recovery mechanism, and the inserted replica will also endure such phase. This warm-up phase occurs during the first 30 s of the experiment as presented in Fig. 5.2. Second, only a small interval is needed between insertions and removals, since

Component	Qty.	Description	Component	Qty.	Description
Adaptation Manager	1	Dell PowerEdge 850 Intel Pentium 4 CPU 2.80GHz 1 single-core, HT 2.8 GHz / 1 MB L2 2 GB RAM / DIMM 533MHz 2 x Gigabit Eth. Hard disk 80 GB / SCSI	Client (YCSB)	1	Dell PowerEdge R410 Intel Xeon E5520 2 quad-core, HT 2.27 GHz / 1 MB L2 / 8 MB L3 32 GB / DIMM 1066 MHz 2 x Gigabit Eth. Hard disk 146 GB / SCSI
Client (WS-Test)	5		Service Gateway	1	
Cloud RM	3		Physical Cloud Host	6	

Table 5.1: Hardware environment.

the service reconfigures quickly. Third, the service latency increases 20- to 30-fold during recoveries, but throughput (operations/s) never falls to zero.

Impact on a BFT stateful key-value store. Our second test considers a BFT key-value store based on state machine replication. The service group is also composed of 4 large replicas, but it tolerates only 1 arbitrary fault, respecting the $3f + 1$ minimum required by BFT-SMaRt. Fig. 5.3 shows the resulting latencies with and without recovery, regarding (a) PUT and (b) GET operations. The entire experiment took 800 s to finish.



(a) PUT operations

(b) GET operations

Figure 5.3: Impact on a BFT stateful key-value store. Note that the y-axis is in logarithmic scale.

We are able to show the impact of a recovery round on service latency by keeping the rate of requests constant at 1000 operations/s. In this graph, each replica recovery is divided into the insertion of a new replica (marked with "+R") and the removal of an old replica (marked with "-R"). Removing the group leader is marked with "-L". The average latency of PUT operations without recovery was 3.69 ms, whereas with recovery it was 4.15 ms (a difference of 12.52%). Regarding GET operations, the average latency without recovery was 0.79 ms, whereas with recovery was 0.82 ms (a difference of 3.33%).

We draw attention to six aspects of these results. First, the service in question also goes through a warm-up phase during the first 45 s of the experiment. Second, the service needs a bigger interval between insertion and removal than the previous case because a state transfer occurs when inserting a new replica. Third, the service loses almost one third of its capacity on each insertion, which takes more time than in the stateless case. Fourth, the service stops processing during a few seconds (starting at 760 s in Fig. 6.5(a)) when the leader leaves the group. This unavailability while electing a new leader cannot be avoided, since the system is unable to order requests during leader changes. Fifth, client requests sent during this period are queued and answered as soon as the new leader is elected. Finally, GET operations do not suffer the same impact on recovering replicas as PUT operations do because GET operations are executed without being totally ordered across replicas, whereas PUT operations are ordered by BFT-SMaRt protocol.

5.5.3 Scale-out and Scale-in

Horizontal scalability is the ability of increasing or reducing the number of service instances to follow demands of clients. In this experiment, we insert and remove replicas from a stateless service group to adapt the service capacity. The resulting latencies are presented in Fig. 5.4. The entire experiment took 1800 s to finish, and the stateless service processed almost 4 million

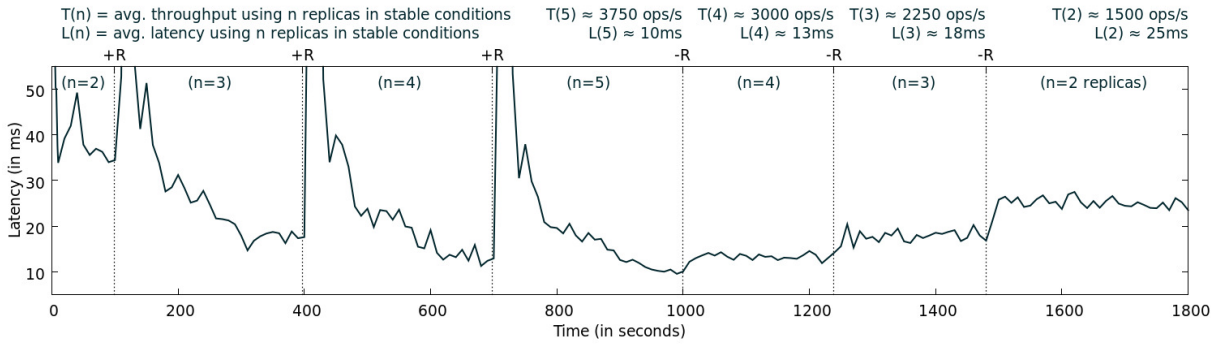


Figure 5.4: Horizontal scalability test.

client requests, resulting in an average of 2220 operations/s. Each adaptation is either composed of a replica insertion ("+R") or removal ("-R").

Since all replicas are identical, we consider that each replica insertion/removal in the group can theoretically improve/reduce the service throughput by $1/n$, where n is the number of replicas running the service before the adaptation request.

The service group was initially composed of 2 small (S) replicas, which means a capacity of processing 1500 operations/s. Near the 100 s mark, the first adaptation was performed, a replica insertion, which decreased the service latency from 30 ms to 18 ms. Other replica insertions were performed near the 400 s and 700 s marks, increasing the service group to 4, and to 5 replicas, and decreasing the service latency to 13 ms, and to 10 ms, respectively. The service achieved its peak performance with 5 replicas near the 900 s mark, and started decreasing the number of replicas with removals near the 1000 s, 1240 s and 1480 s, until when increases the service latency to 25 ms using 2 replicas.

Dynamically adapting the number of replicas can be performed reactively. A comparison between the service capacity and the current rate of client requests can determine if the service needs more or fewer replicas. In case of large differences, the system could insert or remove multiple replicas in the same adaptation request. Such rapid elasticity can adapt better to large peaks and troughs of client requests. We maintained the client request rate above the service capacity to obtain the highest number of processed operations on each second.

The entire experiment would cost $0.200 on Amazon EC2 [Ama06] considering a static approach (using 5 small replicas), while with the dynamic approach it would cost $0.136. Thus, if this workload is repeated continuously, the dynamic approach could provide a monetary gain of 53%, which is equivalent to $1120 per year.

5.5.4 Scale-up and Scale-down

Vertical scalability is achieved through upgrade and downgrade procedures to adjust the service capacity to client's demands. It avoids the disadvantages of increasing the number of replicas [AEMGG+05b], since it maintains the number of replicas after a service adaptation.

In this experiment, we scale-up and -down the replicas of our BFT key-value store, during 8000 s. Each upgrade or downgrade operation is composed of 4 replica replacements in a chain. The service group comprises 4 initially small replicas (4S mark). Fig. 5.5 shows the resulting latencies during the entire experiment, where each "Upgrading" and "Downgrading" mark indicates a scale-up and scale-down, respectively. We also present on top of this figure

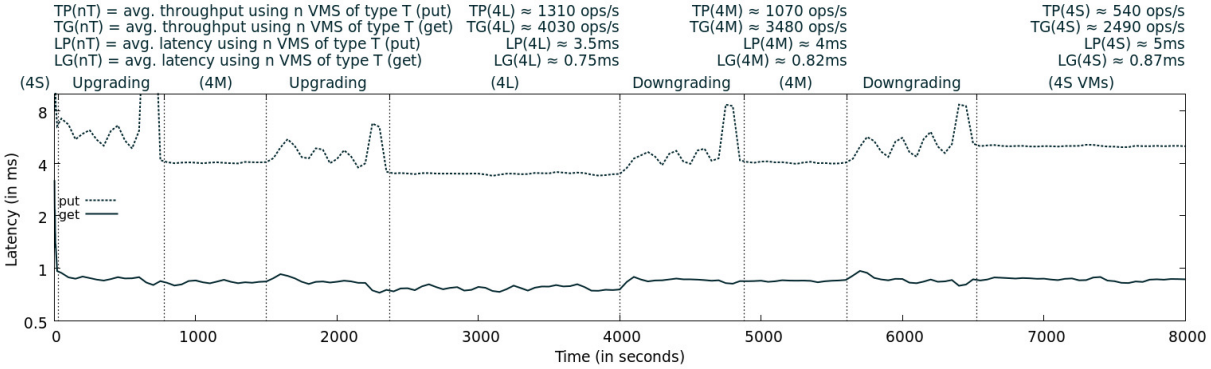


Figure 5.5: Vertical scalability test. Note that y-axis is in logarithmic scale.

the "(4S)", "(4M)" and "(4L)" marks, indicating the quantity and type of VMs used on the entire service group between the previous and the next marks, as well as the average latency and number of operations per second that each configuration is able to process.

The first adaptation was an upgrade from small (S) to medium (M) VMs, which reduced PUT latency from near 6 ms to 4 ms and GET latency from almost 0.9 ms to 0.82 ms. The second round was an upgrade to large (L) VMs. This reduced the PUT latency from 4 ms to 3.5 ms and the GET latency from 0.82 ms to 0.75 ms. Later, we performed downgrades to the service (from large to medium and from medium to small), which reduced the performance and increased the PUT latency to almost 5 ms and the GET latency to 0.87 ms.

The entire experiment would cost $2.84 on Amazon EC2 [Ama06] considering the static approach (using 4 large replicas), while the dynamic approach would cost $1.68. This can be translated into an economical gain of 32%, the equivalent to $4559 per year.

5.6 Related Work

Dynamic resource provisioning is a core functionality in cloud environments. Previous work [BGC11] has studied the basic mechanisms to provide resources given a set of SLAs that define the user's requirements. In our work, cloud managers provide resources based on requests sent by the adaptation manager, for allocating and releasing resources.

The adaptation manager is responsible for performing dynamic adaptations in arbitrary service components. These adaptations are defined by an adaptation heuristic. There are several proposals for this kind of component [BAP07, DPC10, Gar04], which normally follow the "monitor-analyse-plan-execute" adaptation loop [KC03]. Their focus is mostly on preparing decision heuristics, not on executing the dynamic adaptations. Such heuristics are based mainly on performance aspects, whereas our work is concerned with performance and dependability aspects. One of our main goals is to maintain the service trustworthiness level during the entire mission time, even in the presence of faults. As none of the aforementioned papers were concerned with economy aspects, none of them releases or migrate over-provisioned resources to save money [YAK11].

Regarding the results presented in these works, only Rainbow [Gar04] demonstrates the throughput of a stateless web service running over their adaptation system. However, it does not discuss the impact caused by adaptations in the service provisioning. In our evaluation, we demonstrate the impact of replacing an entire group of service replicas, in a stateless web

service, and additionally, in a stateful BFT key-value store.

Regarding the execution step of dynamic adaptation, only Dynaco [BAP07] describes the resource allocation process and executes it using grid resource managers. FITCH is prepared to allow the execution of dynamic adaptations using multiple cloud resource managers. As adaptation heuristics is not the focus of this chapter, we discussed some reactive opportunities, but implemented only time-based proactive heuristics. In the same way, proactive recovery is essential in order to maintain availability of replicated systems in the presence of faults [CL02, DPSP+11, RK07, SNV05, SBC+10]. An important difference to our work is that these systems do not consider the opportunities for dynamic management and elasticity as given in cloud environments.

Dynamic reconfiguration has been considered in previous work on group communication systems [CHS01] and reconfigurable replicated state machines [LMZ10b, LAB+06]. These approaches are orthogonal to our contribution, which is a system architecture that allows taking advantage of a dynamic cloud infrastructure for such reconfigurations.

5.7 Conclusions

Replicated services do not take advantage from the dynamism of cloud resource provisioning to adapt to real-world changing conditions. However, there are opportunities to improve these services in terms of performance, dependability and cost-efficiency if such cloud capabilities are used.

In this chapter, we described and evaluated FITCH, a novel infrastructure to support the dynamic adaptation of replicated services in cloud environments. Our architecture is based on well-understood architectural principles [Ver02] and can be implemented in current data centre architectures [HB09] and cloud platforms with minimal changes. The three basic adaptation operations supported by FITCH – add, remove and replace a replica – were enough to perform all adaptation of interest.

We validated FITCH by implementing two representative services: a crash fault-tolerant web service and a BFT key-value store. We show that it is possible to augment the dependability of such services through proactive recovery with minimal impact on their performance. Moreover, the use of FITCH allows both services to adapt to different workloads through scale-up/down and scale-out/in techniques.

Chapter 6

A Byzantine Fault-Tolerant Transactional Database

Chapter Authors:
Marcel Santos and Alysson Bessani (FFCUL).

6.1 Introduction

The purpose and functionality of most computer systems is the storage and manipulation of information. A wide variety of systems like business applications, health care providers, financial and government institutions uses computer systems to store and access information.

To manage data and provide access to information, database systems were created decades ago. Database management systems (DBMS) are systems that manage and provide tools to access data. There are lots of database vendors that provide DBMS. Aspects like data integrity, availability and performance can vary from vendor to vendor.

Database vendors may provide mechanisms like replication and periodic backups to secure data but these mechanisms may not prevent data loss or corruption. Threats like software bugs, hardware failures and intrusions can modify data or cause loss of data before it is securely stored in durable media. Even after data is stored, it can yet be corrupted or lost.

To address resilience in database systems an alternative is to use active replication. In summary, data is forwarded to several databases and only when a predefined number of participants confirm the result of the operation the result is returned to the client.

State machine replication [CL02] (SMR) is a fault tolerance protocol for active replication. It assumes that data is replicated across several participants, called replicas. The replicas start with the same state and execute operations that change the application state in the same order. Operations are ordered through a sequence of messages exchanged between replicas. The system will have a consistent state even in the presence of a predefined number of faulty replicas. SMR can be implemented to tolerate Byzantine faults using $3f + 1$ replicas to tolerate f faults. In this case, a replica can be faulty not only due to crash or delay to process the requests, but also due to software and hardware bugs that make it return arbitrary results and corrupt its state.

Our goal is to provide a database replication middleware - SteelDB - that works on top of BFT-SMaRt [BSA], a Byzantine Fault Tolerant SMR protocol. This middleware provides resilience in the presence of faulty replicas. Some of the ideas used in SteelDB were defined by Byzantium [GRP11], a database replication middleware built on top of PBFT [CL02], the first efficient BFT-SMR protocol.

6.2 Byzantium

Byzantium is a database replication middleware tolerant to Byzantine faults. It uses PBFT as the replication library and requires $3f + 1$ replicas to tolerate f faults. Byzantium architecture is presented in figure 6.1.

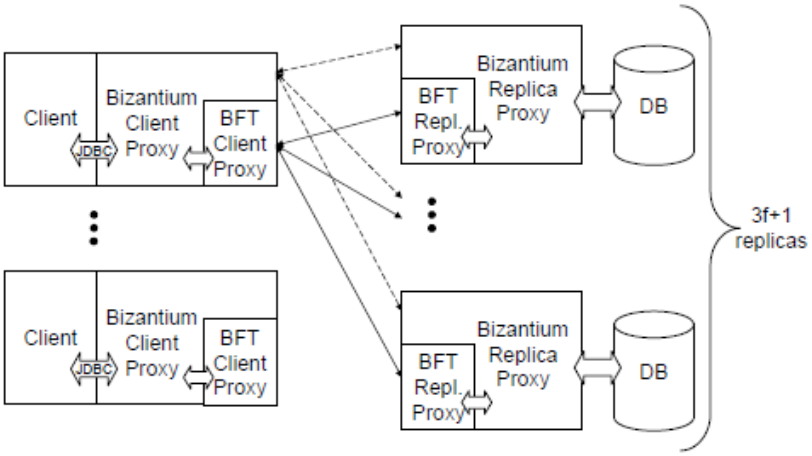


Figure 6.1: The Byzantium architecture.

The state machine replication protocol requires that messages which change the state of the application to be totally ordered across replicas.

Byzantium considers all operations executed against databases to run inside transactions. Transactions are group of operations that holds ACID properties (atomicity, consistency, isolation and durability). When using transactions, the state of the database is persisted only when the transaction is confirmed, a process called commit.

To increase concurrent execution of transactions, Byzantium assumes that DBMS implementation provides snapshot isolation. Snapshot isolation, also known as multi version concurrency control is a technique to control concurrent access to data without the use of table locks. Version numbers are assigned to data objects when they are read or modified. Before commit the database manager verifies if the version of the object being written is the same version as when it was opened.

Transactions in Byzantium are flagged as read-write or read-only. By the time a transaction is created it is considered read-only. Transactions will be promoted to read-write only when the first write message is received.

To reduce the cost of replication, operations received in read-only transactions are sent only to $f + 1$ replicas. The Byzantium client will wait for f replies before return the result to the client. If the remaining replies doesn't match the f received the protocol will try to execute the operations in the other replicas to validate the result.

Operations that occur in a read-write transaction are sent to all replicas but only executed in one of them, called master. This way the client won't need to wait for a reply quorum to confirm the operation. During the processing of a transaction, the client will keep a list of operations sent and results returned by the master replica. By the time of commit the client will request a commit and will send the list of operations and results. The replicas then will verify if the operations match. If operations matches, the replicas will execute all operations and compare the results with results informed from the master. If results match the commit will

be confirmed. Otherwise, the transaction will be rolled back and the client will be informed, so that it can proceed with a suspect master operation.

Byzantium defines also a suspect master protocol to change a faulty master due to crash or Byzantine behavior. When a client notices that the master returned incorrect results or took longer than a predefined timeout to return results, it will send a suspect master message to the replicas. The replicas will try then to confirm with the master if it received the messages from the client. In the case of no success, the replicas will define the replica with the next id as the master and will inform the client about the new master. Open transactions will be rolled back and the clients will be informed about the rollback.

Byzantium defines two versions of the protocol, called single-master and multi-master. In the single-master version, all transactions considers the same replica to be the master. In the multi-master protocol, at the beginning of a transaction one of the replicas is randomly selected to be the master replica. In that case, each transaction will have a master and this master can be different from other transactions. The single-master performs better in a read-write dominated workload while the multi-master version performs better in a read-only dominated workload.

6.3 SteelDB

To provide resilience for the database layer of work package 3.2 we developed SteelDB, a middleware to replicate data across multiple replicas, assuming that a predefined number of replicas can be faulty. SteelDB is a JDBC implementation that requires no changes in client implementation to be used. It also borrows some ideas from Byzantium like the optimistic execution of transactions on master replicas and master change protocol. We have implemented only the single-master protocol of Byzantium. The use case by work package 3.2 assumes that clients are located in the same infrastructure so there will be no advantage to have several different masters. Having a single master in a replica in the same infrastructure as the client would yield faster response times than a master in a external cloud. Also to implement the state transfer protocol in the presence of multiple masters would add increase the number of scenarios to be considered and be very time consuming.

Steel DB uses BFT-SMART (see Chapter 2) as its state machine replication protocol. BFT-SMART is an efficient state machine replication programming library that tolerates crash and Byzantine faults. It is implemented in Java and includes several protocols like state transfer, leader change and reconfiguration to recover from faults and increase performance on different workloads. To have JDBC as a client of BFT-SMART and replicate data to the server replicas, we had to implement BFT-SMART interfaces for clients and servers. The architecture of SteelDB is presented in Figure 6.2.

The clients of SteelDB implement the JDBC specification to invoke operations in BFT-SMART service proxy. We take advantage of the use of transactions to reduce the number of messages exchanged between clients and replicas. Unlike in Byzantium, when an operation is inside a transaction, the client will send it only to the master replica. The master will execute the operation and return the result to the client. By the commit time the client will send the list of operations executed and responses received during the transaction to all replicas. The replicas will execute the operations and check the results against the results returned from the master.

The sequence of messages executed by the replicas is presented in the Figure 6.3:

1. The replica receives an array of bytes through BFT-SMART interface.

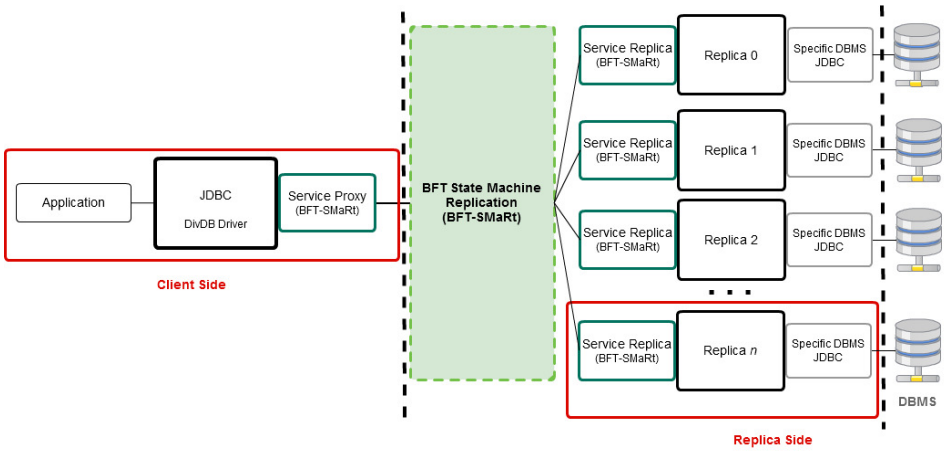


Figure 6.2: The SteelDB architecture.

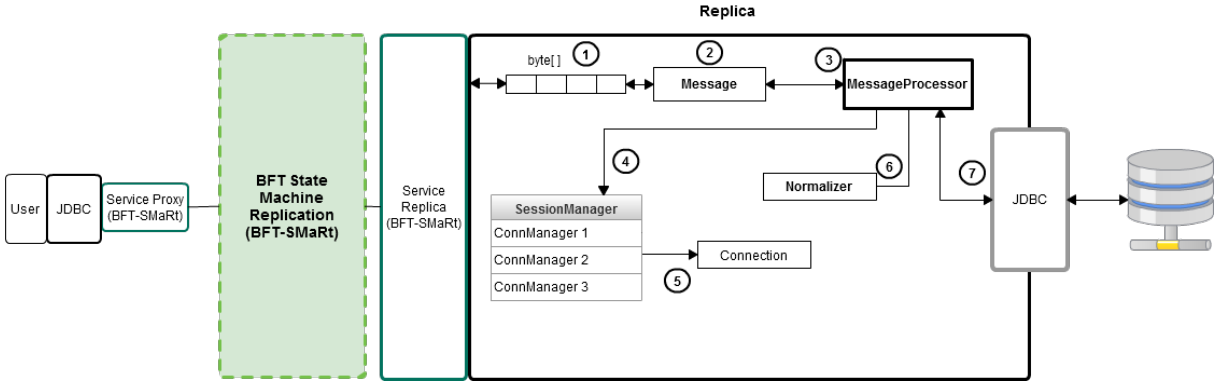


Figure 6.3: Work-flow inside a replica.

2. The array of bytes is then transformed in a Message object. This object is understandable by the replica, unlike the array of bytes.

3. The message is then processed by the MessageProcessor according to the kind of request.

4. If the message is a login operation, the login is executed and the connection stored in the SessionManager.

5. SessionManager returns the connection to execute the operation.

6. When SteelDB uses DBMS from different vendors, the message has to be normalized to allow the comparison of results from replicas.

7. The operation is executed against the JDBC interface. If this is the master replica the result is returned to the client. If not, each operation sent with the commit message is checked against the results from master before confirming the commit.

We will describe next the ideas presented on SteelDB and the issues we had during its implementation.

6.3.1 Fifo Order

All previous services developed over BFT-SMaRt supported simple read, write or read-write operations, without any support for ACID transactions. In this scenario, there was no problem if some unordered read operation arrived out of order in f replicas. This happens because the client expects a number of replicas to execute operations and return matching results before sending its next operation. The problem is, with a transactional protocol, if an operation within a transaction arrives at some replica before the transaction BEGIN (or the database connection is established), it will be rejected, and the client may observe problems. The reason for this problem is that BFT-SMaRt either considers total order requests or unordered requests, there is no mechanism ensuring FIFO order, as required in Byzantium. To deal with this limitation without modifying BFT-SMaRt we devised an ordering layer that uses application-level sequence numbers (defined in our JDBC driver and verified in the replica' service) and locks to ensure messages are processed in the order that they are sent by a client. After several tests we discovered that this would not work without changing the internal handling of requests in BFT-SMaRt.

We found that it would be easier to change the protocol to optimistically execute operations inside transactions only in the master replica and send them to the other replicas only before commit. This removed the requirement of execute clients messages in FIFO order.

6.3.2 JDBC specification

The transaction management in JDBC is slightly different from the way that a DBMS operates. Usually, the sequence of steps to process a transaction in a database includes: open a connection, begin a transaction, executes the operations and commit or rollback the transaction. The description of Byzantium algorithm follows this sequence of operations.

JDBC manages the transaction boundaries transparently. When a user opens a connection to the database, the connection is opened in auto-commit mode. This means that all operations will be reflected in the database immediately, without need for a commit operation. If the client needs to create a transaction, it has to change the connection auto-commit flag to false. This has the same effect as issuing a begin command to the database. Operations are executed in the database and the transaction is finished by a commit or a rollback command. After a commit, or a rollback, the auto-commit flag is not changed, which means that the connection is still not auto-commit, so, one transaction was committed and another started. A client can also finish a transaction by setting again the auto-commit flag to true. It will commit the current transaction before the change.

We had to manage client operations to store the auto-commit flag for all operations. Byzantium also defines that transactions shouldn't be considered read-write before the first update operation is issued. To implement that we needed to add another flag to each connection to define if it has a read-only or read-write transaction. By the time a transaction is promoted to read-write, all operations are executed only in the master replica. In the other replicas operations are executed only before commit.

6.3.3 State transfer

State machine replication assumes that a replication system will make progress even in the presence of up to f faulty replicas. This means that situations may occur when up to f replicas are not in the same state of the others. Also there may be times when a replica crashes and

recovers after some time. As the system continues to make progress, the recovered or late replica may not have an up to date copy of the state, so it has to ask other replicas to provide the state. This recovery process is managed by the state transfer protocol.

Without the use of a state transfer protocol, after f replicas are compromised, the system is vulnerable to attacks or crashes. At that point if any additional faults occur, the integrity of the system cannot be confirmed as there is not a presence of a minimum quorum of participants to validate operations.

The state transfer protocol is started when a replica receives a sequence of operations and realizes that the operations sequence numbers are higher than the last it executed, or that it didn't execute operations at all.

The replica will send a request to other replicas asking for the application state. One replica is defined to send the state and the others will send digests of the state. After receiving the state and digests, the replica can compare the data to find if the state is correct. If confirmed, the state is installed and the replica continues to process the remaining requests. The state transfer protocol recovers a faulty replica, enabling it to process requests again. That process will reduce the window of vulnerability of the system.

Byzantium defined an optimization where replicas would store operations to be executed before commit. We tried to implement this protocol, but it added a lot of complexity to the state transfer protocol. We had to manage the state of the application plus partial data that was not stored in the database but couldn't be ignored. This data would need to be executed in the new replica after the state was restored.

The change that we made in the protocol to only execute operations in the master allowed us to only transfer the state from the database of transactions that were commited. BFT-SMART state transfer protocol defines a strategy to take copies of the application state to be transfered to replicas that may request it during the protocol execution. This stratagy comprises the log of all operations that changes the state of the application. To bound the size of the log data, in pre defined periods, the application state is saved in a serialized format called checkpoint and the log is erased. In SteelDB we ask the DBMS to generate a dump of the database information during checkpoint periods. Together with the dump we store the information about open connections to be restored in the new replicas. Operations are logged only during commit time.

When a recovering replica asks the state from other replicas it receives the dump of the database together with information about connections. It can then restore the database, open the connections and execute the logged operations to update itself to the state it received the first request after start.

6.3.4 Master change

Byzantium defines a master change protocol. When the master replica takes more then a predefined timeout to process a request, the client informs the other replicas about the presence of a faulty master in a master change request. Together with this request, the client sends the operations executed so far in the current transaction. When the replicas receive the request for a master change operation, they will update its master id to the new one. The new master replica will execute the requests informed by the client and reply with the results.

To prevent a malicious client from request several master changes, the replicas logs the requests sents from clients. If a client try to request multiple master changes in less then a predefined time, the requests are ignored.

After a master change, clients will not know about the new master until the execution of the next operation. If the next operation is sent only to the old master, it will fail as the old

master is offline. When the client request the master change, replicas will know that the change was already executed and the client will be informed. If instead of send a message only to the master, the client send a message to all replicas (commit message or message in an autocommit connection) the operation will be executed and the replicas will inform the new master together with the reply to the client.

6.3.5 Optimizations

Byzantium defines optimizations in the protocol to increase performance in the presence of read-only dominated workloads.

Messages executed on read-only transactions can be executed speculatively in only two replicas and the first result is returned to client. Only by the time of commit the second result is compared. This works when the operation is executed inside a transaction. If the connection is a auto-commit connection, the message has yet to be validated by a quorum of replicas, as there is no commit time to invalidate the response.

As described before, the storage of operations in replicas before commit time would increase the complexity of the state transfer protocol of BFT-SMART. We decided to instead execute operations only in the master replica and execute the whole transaction before commit in the other replicas.

6.4 Evaluation

To evaluate the performance of SteelDB we used a widely known DBMS benchmarking tool, TPC-C [tpc12]. We executed tests in different scenarios with different database configurations and compared the results.

We first ran tests with TPC-C executing operations against a single database. In a second step we ran TPC-C connected to SteelDB and tested data being replicated to DBMS from different vendors.

6.4.1 TPC-C

TPC-C is a benchmarking tool that simulates Online Transaction Processing workloads (OLTP) workloads. OLTP refers to a class of systems which has intensive transaction processing for writing retrieving information.

TPC-C simulates a business application to manage the creation and processing of orders across several departments of a company. The database schema of TPC-C is displayed in figure 6.4.

The numbers in the entity blocks represent the number or rows contained in each table. These numbers are scaled by W, which is the number of warehouses, to simulate database scaling. As the number of W increases, the database will accept more clients terminals connected to the database. For each warehouse defined, the database will accept connections from 10 client terminals.

TPC-C defines five different type of transactions that have different workloads and operation types:

- New-Order transaction: mid-weight read-write transaction;
- Payment transaction: light weight read-write transaction;

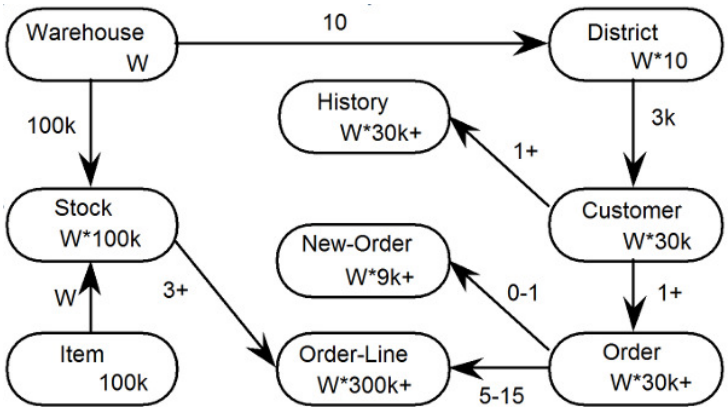


Figure 6.4: TPC-C database schema.

- Order-status: mid-weight read-only transaction;
- Delivery transaction: simulates execution of ten new order transactions. It is supposed to be executed in deferred mode, unlike the other transactions;
- Stock-level transaction: heavy weight read-only transaction.

Several benchmark users [VBLM07, GRP11] measure how many transactions per minute-C (tpmC) the database can execute. That value represents how many New-Order transactions per minute a system generates while the system is executing the other four transaction types. We will use that measure in our experiments.

We couldn't use H2 database in the tests because of incompatibility issues with TPC-C queries. We were not able to create the tables needed by TPC-C and also could not load the data to run the tests. Despite that, we ran the tests with different vendors to evaluate the performance of SteelDB compared to direct access to databases.

We do not support the execution of SteelDB protocol with different database vendors. One step of the state transfer protocol includes the request of a database dump to the DBMS. Different vendors provide different dialects and structures for database dump. As the state transfer protocol requires the dumps to be equal, we would not be able to compare dumps from different vendors. We did though implement a module to normalize queries. That module allowed us to run the benchmark against a diverse configuration of databases.

6.4.2 Test environment

We ran TPC-C benchmarking tool against DBMS from different vendors. We used MySQL [mys12], PostgreSQL [pos12], HSQLDB [hyp12] and FirebirdSQL [fir12] and compared the results.

Tests were executed in a cluster of machines, each one with an Intel Xeon E5520 2.27 GHz processor, 32GB of RAM memory, 15K RPM SCSI disk and a Gigabit Ethernet network interface. Machines were running the Ubuntu 10.04 Server 64-bits operating system, with the kernel version 2.6.32. The Java Virtual Machine used was OpenJDK Runtime Environment version `1.6.0_18`. The versions of the DBMS used were MySQL 5.1, PostgreSQL 9.1, HyperSQL 2.2.8 and Firebird SuperClassic 2.5.1.

To measure transaction processing and disregard side effects as JIT compilation time we made slight modifications to the benchmark:

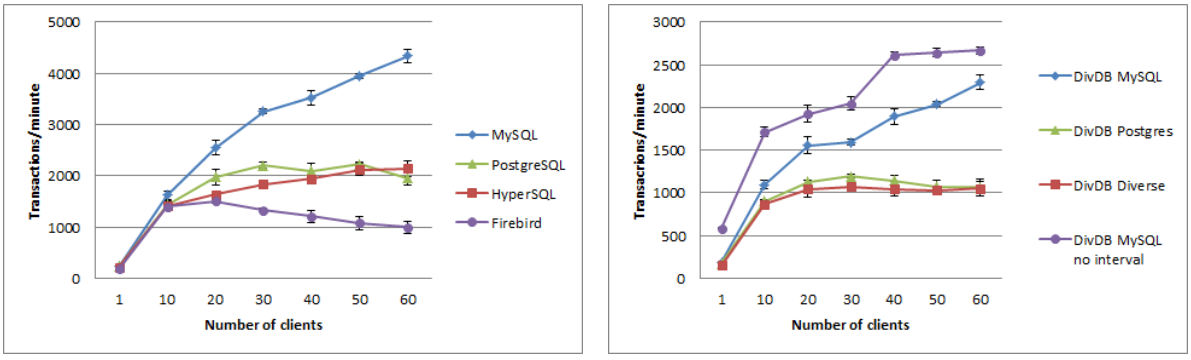
- added a 1 minute warm-up phase before starting the performance measurements;
- allowed clients to execute on different machines (maximum of 30 clients per machine);
- added a inter-transaction arrival time of 200 ms. In other words, we added an interval of 200 ms between the execution of transactions in each connection.

We used a benchmark workload configuration equivalent to the workload used in Byzantium experiments. The workload was composed as follows:

- 50% read transactions: with 25% for both Stock-Level and Order-Status.
- 50% write transactions: with 27% for New-Order, 21% for Payment and 2% for Delivery.

6.4.3 Test results

We first ran TPC-C benchmark against a single database with four different DBMS implementations. As we can see in Figure 6.5(a) MySQL has the best performance for the workload we set up. We tested it with sixty clients and still didn't get to a point where the throughput was saturated and the latency increased. PostgreSQL and HyperSQL had similar results with a peak throughput between ten and twenty clients, with constant values until sixty clients. Firebird had its peak throughput with ten clients and after that the performance decreased when new clients were introduced. We searched for the reasons for that strange behavior and we discovered that as we increased the number of clients the DBMS access to disk also increased, crippling the performance of the system. We decided not to spend further time on that issue, as it is not a purpose of this work to find performance issues of a specific DBMS vendor.



(a) Standalone DBMS.

(b) SteelDB variants.

Figure 6.5: TPC-C results standalone DBMS and SteelDB variants.

After run TPC-C against single databases, we changed the configuration to use SteelDB as the database driver and have it connected to four different configurations:

- SteelDB Mysql: four instances of MySQL;
- SteelDB Postgres: four instances of PostgreSQL;
- SteelDB Diverse: four instances using different DBMS: MySQL, PostgreSQL, HyperSQL and Firebird;

- SteelDB MySQL no interval: four instances of MySQL without the interval of 200ms between transactions.

Results are displayed in Figure 6.5(b). We can see that the best performance was achieved with MySQL without the interval between transactions. Even then we can notice that the throughput is about 50% of the value obtained with the standalone execution of TPC-C. This is justified by the operations being executed speculatively in the master replica. By the time of commit the operations are replicated to the other replicas. The client have to wait for operations to be executed and checked against operations executed on the master. This means that the client will have to wait for the execution in to steps, one from the master and one from the other replicas. The second slower replica will define the time to commit the operations, as clients will consider only the first replies it receives, disregarding f.

6.5 Lessons learned

During implementation of SteelDB we found issues that were not expected by the time we designed it. We will summarize now some of this issues and the lessons we learned during development and tests.

6.5.1 Use case complexity

Until the work on SteelDB we had implemented several applications on top of BFT-SMaRt. These use cases included simple key-value stores, tuple spaces and file systems. Each use case contained its level of complexity, but all of them differs from a database in the sense that the application was created by ourselves, with characteristics designed to address our needs. During the implementation of SteelDB we had to define methods to receive metadata about tables, columns and data types. We had to set databases to the same timezone to have consistent values. We ran several tests with state transfer to find out that dumps can have queries in different positions and therefore cannot be compared.

In summary, implementing a use case from top to bottom is less prone to surprises than adapting an existing application to the model we had.

6.5.2 DBMS maturity

By the time we planned the replication environment, it was defined that the database to be used would be H2 database for reasons like ease of installation and management of database instances. As we worked in the implementation of the driver we discovered that features that H2 database provides, namely snapshot isolation were not quite the way it should be. The definition of snapshot isolation defines that version numbers should be attributed to data to avoid the use of locks. After analysis on documentation and forums we found out that H2 uses locks for row with lock timeout to manage the concurrent execution of updates. We had then to change our tests so that we not had two different clients changing the same row.

We also needed to make dumps of the database to be used in state transfer. We found out that the dump tools provided didn't include options to have table structures in the dump. MySQL for instance has a large set of options that can be used when taking a dump. It is even possible to include queries to modify data when taking a dump.

When choosing between database vendors, features like ease of use, complexity of management of databases and language are important. But it is also important to check how mature a

DBMS is. Some of the functionality needed may not be available yet. And even some that are claimed to exist may not be implemented as they were defined.

6.5.3 Database specific issues

Databases have mechanisms to transfer the state from one database to another. Most or all database vendors provide dump operations to generate text files with structure and data of the whole database in sequences of queries that can be executed in the new database to restore the state. Although it is a common functionality in database systems, the idiom in which the data is written in the dump files can differ from vendor to vendor. The differences in the structured query language from one database vendor to another are referred to as database dialects. As we use dumps and hashes of data to compare queries results, even a single character that differs from a data set to another can invalidate the results, even if they are equivalent. Because of that, it is impossible to use DBMS from different vendors using the middleware we created. To fix that we would need to extend the normalizer to translate every result from databases to a common dialect. That work would require a lot of effort as there are a huge number of database vendors and the differences among dialects are quite extensive.

Another problem we found was database specific behaviors regarded field types. We had problems with date values that were written in the database and when queried, different replicas returned different results. After days of tests with queries we found out that the database driver we used considers the timezone in which a database is based. Depending on the timezone, the database changed data to reflect the differences between timezones. We couldn't then validate the results. Only when we forced the databases to operate in the same timezone we were able to obtain consistent results.

6.6 Final remarks

The work to implement a client of BFT-SMART to replicate data across multiple database instances was more complex than we thought at first. Databases have several features designed over decades to attend to patterns, clients needs and compatibility with legacy systems. These features make such use case complex to be implemented in its full potential. We chose to implement features that were needed by the use cases we had on work package 3.2.

The tests on SteelDB were complex because the use case comprises the large set of scenarios with multiple clients creating transactions, executing queries and comparing results. The tests would not be possible without the help from our partners from work package 3.2, who provided us several test suites to validate the operation of the protocol.

Despite the issues and problems we had along the way, we could implement a JDBC driver that passed the tests we had in different scenarios and provides resilience for database replication.

Chapter 7

The C2FS Cloud-of-Clouds File System

Chapter Authors:
Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Marcelo Pasin, Miguel Correia and Paulo Veríssimo (FFCUL).

7.1 Introduction

A recent survey shows that backup, archiving and collaboration tools are among the top uses of cloud services by companies [SUR]. All of these uses involve storage services like Amazon S3, Dropbox, Apple iClouds, or Microsoft SkyDrive. The reasons for the popularity of these services are their ubiquitous accessibility, scalability, pay-per-use model, and simplicity of use for sharing data.

On the other hand, there is a certain reluctance to place too much trust on cloud storage service providers (CSSPs), given a history of various kinds of problems: unavailability due to outages [Bec09, Cho12], vendor lock-in [ALPW10], and data corruption due to bugs and errors [AWS11, Ham12], malicious insiders or external cyber-attacks [Clo10, Dek12]. Part of these threats may be overcome by solutions relying on client-side mechanisms that provide end-to-end guarantees [MSL+10]. However, some of the threats above are common-mode, i.e., they will persist even with those protection mechanisms, in the case of a single CSSP – unavailability or lock-in being the most obvious examples. In consequence, other recent works have proposed the use of a *cloud-of-clouds*, i.e., a combination of cloud storage services from different providers [ALPW10, BCE+12, BCQ+11]. The basic idea is to store data redundantly under the expectation that such providers will have independent failure modes. Whatever protection mechanisms are used, they will no longer depend on a single provider, giving a broad threat coverage.

Cloud-of-clouds storage systems have thus been showing promise. It has in particular been demonstrated that it is possible to build a trusted object storage (a set of registers in distributed computing terminology [Lam86]) over a diverse set of untrusted CSSPs, without incurring prohibitive costs in terms of performance or budget [BCQ+11]. However, the trustworthiness and performance gains are obscured by lesser usability for end service users than say, databases and file systems.

This chapter tries to achieve the best of both worlds, by proposing a *cloud-of-clouds file system*, leveraging the use of multiple clouds for trustworthiness, and simultaneously offering a regular file system interface, as a natural extension, with a stronger semantics, of the basic object storage abstraction of current cloud-of-clouds storage systems [BCE+12, BCQ+11]. Easy integration with existing local applications is promoted by making the API POSIX-like, with the corresponding well-known semantics.

The proposed file system, dubbed *C2FS* (Cloud-of-Clouds File System), leverages almost

30 years of literature on distributed file systems, integrating classical ideas such as the consistency-on-close semantics of AFS [HKM+88] and the separation of data and metadata of NASD [GNA+98], with recent trends like using cloud services as a backplane for storage [DMM+12, VSV12]. In a nutshell, C2FS contributes with the following novel ideas in cloud-backed storage design:

- *Multiple redundant cloud backends:* the backend of C2FS is a set of (unmodified) cloud storage services. All data stored in the clouds is encrypted and encoded for confidentiality and storage-efficiency.
- *Always write / avoid reading:* writing to the cloud is cheap, but reading is expensive [1]. C2FS exploits this behavior for maximum cost-effectiveness: any updates on file contents are pushed to the cloud (besides being written locally), but reads are resolved locally, whenever possible.
- *Modular coordination:* instead of embedding customized locking and metadata functionality, as in most distributed file systems [ABC+02, KBC+00, WBM+06], C2FS uses a modular and fault-tolerant coordination service, which not only maintains metadata information and performs concurrency control, but also assists consistency and sharing management.
- *Consistency anchors:* an innovative principle and mechanism for providing strong consistency guarantees, as opposed to the usual eventual consistency [Vog09], offered by cloud storage services, but unnatural for programmers. We explore the new opportunities offered by a file system, to improve the consistency/performance tradeoff without requiring any modification to the cloud storage services.

The resulting system is a robust cloud-backed file system that uses the local storage plus a set of providers for achieving extreme durability despite a large spectrum of failures and security threats. Besides the specific design, we contribute with a set of generic principles for cloud-backed file system design, re-applicable in further systems with different purposes than ours.

A system like C2FS can be interesting for both individuals and large organizations wanting to explore the benefits of cloud-backed storage. Some example use-cases are: *secure personal file system* – similar to Dropbox, iClouds or SkyDrive, but without requiring complete trust on any single provider; *shared file system for organizations* – cost-effective storage, but maintaining control and privacy of organizations' data; *automatic disaster recovery* – organizations' files stored in C2FS survive disasters not only of their IT but also of individual cloud providers; *collaboration infrastructure* – dependable data-based collaborative applications without running code in the cloud, made easy by the POSIX-like API for sharing files.

We start the remaining of the chapter by discussing the limitations of current cloud-backed file systems and our goals with C2FS (§7.2). The following section (§7.3) describes a general framework for strengthening the consistency of cloud storage services, through the judicious use of stronger consistency services as consistency anchors. This framework is used in the design of C2FS, as described in §7.4, and its implementation, described in §7.5. An extensive evaluation of C2FS is presented in §7.6, just before a discussion of the related work (§7.7) and the conclusions of the chapter (§7.8).

[1] For example, in the Amazon S3, writing is actually free, and reading a GB is more expensive ($0.12 after the first GB/month) than storing data ($0.09 per GB/month). Google Storage's prices are similar.

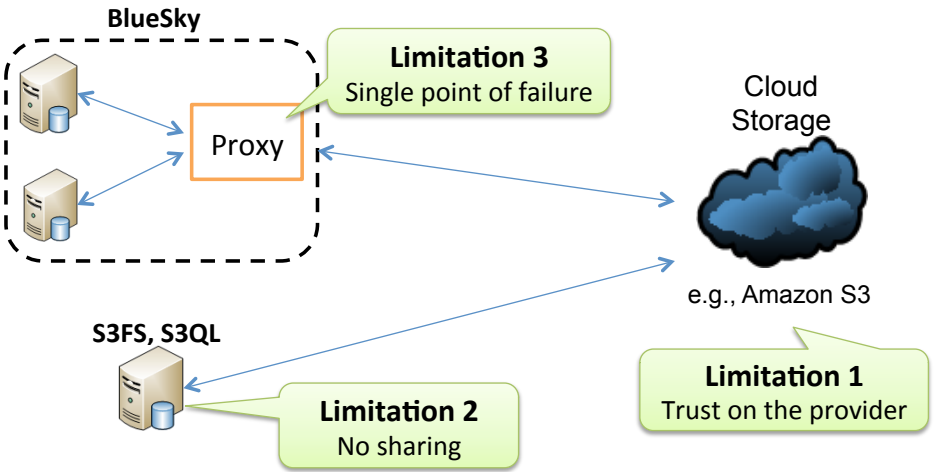


Figure 7.1: Cloud-backed file systems and their limitations.

7.2 Context, Goals and Assumptions

7.2.1 Cloud-backed File Systems

Existing cloud-backed file systems mainly follow one of the two architectural models represented in Figure 7.1. The first model is implemented by BlueSky [VSV12] and several commercial storage gateways (top of the figure). In this approach, a proxy is placed in the network infrastructure of the organization, acting as a file server to the various clients and supporting one or more access protocols, such as NFS and CIFS. The proxy implements the core functionality of the file system, and interacts with the cloud to maintain the file data. File sharing is allowed in a controlled way among clients, as long as they use the same proxy. The main limitations are that the proxy can become a bottleneck and a single point of failure.

The second model is used in some open-source cloud-backed file systems like S3FS [S3F] and S3QL [S3Qa] (bottom of Figure 7.1). These solutions let file system clients access directly the cloud storage services. The absence of a proxy removes the single point of failure, which is positive, but also removes the convenient rendezvous point for synchronization, creating difficulties in supporting file sharing among clients.

A common limitation in both models is the need to trust the cloud storage provider with respect to stored data confidentiality, integrity and availability. Although confidentiality can be easily guaranteed by making clients (or the proxy) encrypt data before sending it to the cloud, the problem is that sharing files in this way requires some key distribution mechanism, which is not trivial to implement in a cloud environment, specially without the proxy. Integrity is already provided in a few systems, such as SUNDR [LKMS04] and Depot [MSL+10], but they need server-side code to run in the cloud storage provider, which is often not allowed by these services (e.g., S3FS and S3QL do not need to execute code in the cloud as they use the Amazon S3's RESTful interface). Availability against cloud provider failures is, to the best of our knowledge, not secured by current cloud-backed file systems.

7.2.2 Goals

C2FS most important goals are to ensure *service availability*, *integrity* and *file content confidentiality* for a cloud-backed file system. C2FS leverages the concept of cloud-of-clouds, ensuring availability in face of cloud failures, by replicating data in multiple providers. More specifically,

C2FS builds on the DepSky read and write protocols for raw storage from multiple untrusted cloud storage providers [BCQ+11] (described in Chapter 4 of TClouds deliverable D2.2.2). These protocols integrate quorums, secret sharing and erasure codes, to implement a Byzantine fault-tolerant register in the cloud, preserving the integrity and confidentiality of the content.

Another goal is to offer *strong consistency* with well-defined file system semantics. In particular, C2FS provides *consistency-on-close semantics* [HKM+88], guaranteeing that *when a file is closed by a user, all updates seen or done by this user will be observed by the rest of the users*. Since storage clouds only have eventual consistency, we resort to a coordination service for maintaining file system metadata and synchronization. We rely on DepSpace [BACF08] for this service, as it ensures linearizability on the operations performed on a deterministic tuple space datastore [Gel85] implemented over a Byzantine fault-tolerant state machine replication.

A last goal is to leverage the scalability promised by the clouds, to support large *numbers of users*, *stored data*, and *numbers of files*. Of course, C2FS is constrained by the limits of the underlying infrastructure, and in particular it is not intended to be a "big data" file system, since data is saved in the cloud and needs to be uploaded/downloaded. Recall that one of the key success factors of big data processing models like MapReduce is that computation and storage happen in the same nodes.

7.2.3 System and Threat Model

C2FS requires a group of n_s cloud storage providers and on a set of n_c computing nodes. The former keeps the file data and the latter executes the coordination service. The same provider can serve both types of resources, but this is optional. We assume that a subset of less than one third of the storage providers or computing nodes can become unavailable, or be subject to Byzantine failures (i.e., $f_s < \frac{n_s}{3}$ and $f_c < \frac{n_c}{3}$). This is the optimal bound required for tolerating Byzantine faults in a replicated system [LSP82]. Moreover, the system can have an undefined number of C2FS clients that may also experience Byzantine failures.

Each C2FS client mounts the file system to enable users to access the storage. To enforce access control restrictions (see §7.4.4), the user needs to have the following credentials: (1) a public and private key pair to authenticate with the coordination service, where the public key is stored in a certificate signed by some trusted certification authority; (2) a set of n_s credentials (e.g., canonical id and password) to access the different storage clouds used by the system. To simplify the discussion, we will abstract these credentials and consider that all of them are available to the user (in practice, the public and private key pair is stored securely in the client machine, and the cloud credentials are obtained from the coordination service).

7.3 Strengthening Cloud Consistency

A key innovation of C2FS is the ability to provide a stronger consistent storage over the eventually-consistent services offered by clouds [Vog09]. Given the recent interest in strengthening eventual consistency in other areas, such as in geo-replication [LFKA11], we describe the general technique here, decoupled from the file system design.

The approach is based in two storage systems, one with limited capacity for maintaining metadata and another to save the data itself. We call the metadata store a *consistency anchor (CA)* and require it to enforce some desired consistency guarantee S (e.g., linearizability [HW90]), while the *storage service* (SS) may only offer eventual consistency. The aim is to provide a composite storage system that satisfies S, even if the data is kept in SS.

WRITE(id, v):	READ(id):
w1: $h \leftarrow \mathsf{H}(v)$	**r1:** $h \leftarrow$ CA.read(id)
w2: SS.write($id\|h, v$)	**r2:** **while** $v = null$ **do**
w3: CA.write(id, h)	**r3:** $v \leftarrow$ SS.read($id\|h$)
	r4: **return** $(h = \mathsf{H}(v))?v : \perp$

Figure 7.2: Algorithm for increasing the consistency of the storage service (SS) using a consistency anchor (CA).

The algorithm for improving consistency is presented in Figure 7.2, and the insight is to anchor the consistency of the resulting storage service on the consistency offered by the CA. For writing, the client starts by calculating a hash of the data object (step w1), and then saves the data in the SS together with its identifier id concatenated with the hash (step w2). Finally, data's identifier and hash are stored in the CA (step w3). One should notice that this mode of operation creates a new version of the data object in every write. Therefore, a garbage collection mechanism is needed to reclaim the storage space of no longer needed versions.

For reading, the client has to obtain the current hash of the data from CA (step r1), and then needs to keep on fetching the data object from the SS until a copy is available (steps r2 and r3). The loop is necessary due to the eventual consistency of the SS — after a write completes, the new hash can be immediately acquired from the CA, but the data is only eventually available in the SS.

In terms of consistency it is easy to see that the combined system will satisfy S. If a data object is updated in the CA, the availability of this last version to other clients is ruled by S. For instance, if the CA satisfies linearizability [HW90], after a writer updates an id with a new hash, any read executed in the CA will either see this hash or a hash produced later on. Furthermore, after a read completes returning some version of the data, no later read will return a previous version of the data.

To tolerate client crashes, the algorithm enforces the property that *a write is complete only if and when the client updates both SS and CA*. If a client changes SS and then crashes, the new version will not be available for reading because the new data hash will not be in the CA, but the old value can still be read. On the other hand, we have a procedure to tolerate missing or wrong writes which also handles misbehaving clients. The while loop in the read procedure exits after a timeout expires (not represented in step r2, for simplicity), thus preventing missing writes to SS from blocking it. Then, the exit value from step r3 is checked against the hash (step r4), preventing wrong values from being returned. In consequence, if e.g., a malicious client writes a hash in the CA and a non-matching data in SS, READ returns with no value.

7.4 C2FS Design

This section explains the design of C2FS. We start by presenting the most relevant design principles, and then we describe the architecture of C2FS and give an overview of the main components. Next, we look in detail into the operation of the C2FS Agent, addressing issues related to caching, consistency and security model.

7.4.1 Design Principles

Pay-per-ownership. In a multi-user cloud-backed file system each client (entity with an account in the clouds) should be charged by the files it owns. This principle has several benefits. First, it gives flexibility to the system's usage model (e.g., different organizations can still share directories paying only for what they create). Second, the shared service is built upon several cloud accounts, allowing to reuse the protection and the isolation already granted by the providers.

Strong consistency. Besides presenting a more familiar storage abstraction (when compared with common cloud interfaces like the RESTful object-based storage), an important added value that a file system can offer is consistency and coherency semantics stronger than the ones ensured by the storage clouds. We achieve this by applying the concept of consistency anchors described in §7.3. Due to this, we opted to design C2FS to support strong consistency by default, supporting other modes of operation, with only eventual guarantees if required.

Legacy preservation. A fundamental requirement of any practical cloud-backed storage system is the use of the clouds as they are. It means that any feature needed from the cloud provider that is not currently supported should be ruled out from any pragmatical design. Accordingly, C2FS does not assume any special feature of computing and storage clouds, requiring only that the clouds provide access to on-demand storage with basic access control lists and well-provisioned VMs connected to the internet.

Multi-versioning. One of the big advantages of having the cloud as a backend is the potentially unlimited scalability. C2FS neither deletes old versions of files nor destroys deleted files, but keeps them in the cloud until a configurable garbage collector removes them. We believe using the unbounded storage capacity of the cloud for keeping old files and traveling in time is a valuable feature for several classes of applications.

Untrusted clouds. Cloud providers can become unavailable due to hardware faults, bugs, disasters, and operator errors. Even more problematic are the rare cases in which these problems corrupt stored data. Storing data in third party clouds may also raise privacy issues, since the data becomes under the control of the provider. For these reasons we avoid trusting any provider individually, and rely instead on distributed trust [SZ05]: each and every piece of data is both read-from and written-into a set of providers, allowing tolerance to unavailability or misbehavior of some of them.

7.4.2 Architecture Overview

Figure 7.3 represents the C2FS architecture with its three main components: the *cloud-of-clouds storage* for saving the file data in a set of cloud storage services; the *coordination service* for managing the metadata and to support synchronization; and the *C2FS Agent* that implements most of C2FS functionality, and corresponds to the file system client mounted at the user machine.

The separation of file metadata from the data allows for parallel access to different file objects, and is already being utilized in individual cloud providers (e.g., [Cal11]). In C2FS, we took this concept further and applied it to a cloud-of-clouds file system. The fact that a distinct service is used for storing metadata gives some flexibility, as we can consider different ways for its deployment depending on the users needs. Although we keep metadata in the cloud in our general architecture, a large organization that uses the system for disaster tolerance and as collaboration infrastructure could distribute the metadata service over its sites. Furthermore, storage clouds support only basic containers and object abstractions for storage, which are ade-

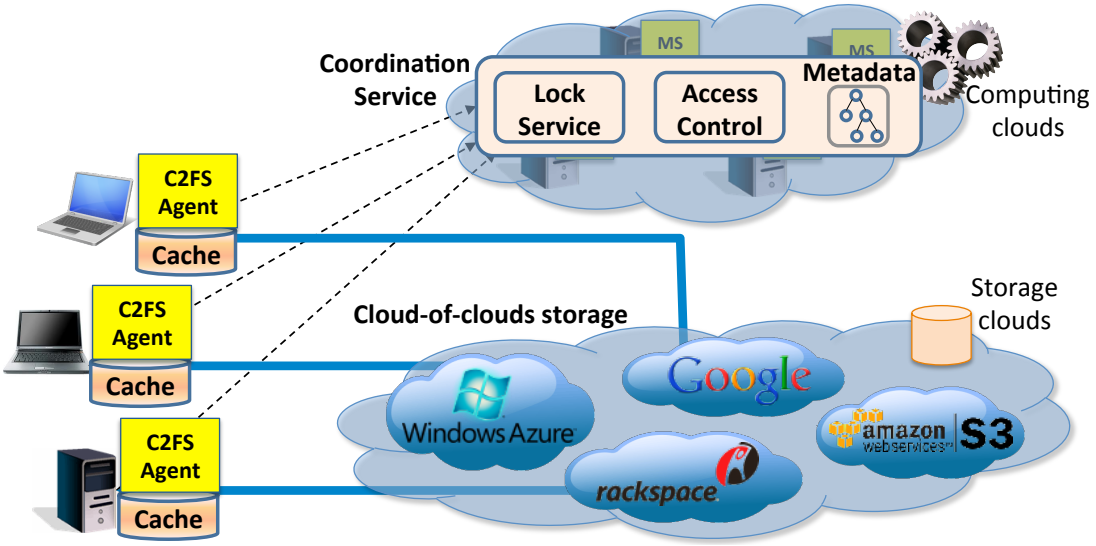


Figure 7.3: C2FS architecture with its three main components.

quate for saving data but not as much to maintain metadata for things like file locks and links.

The metadata storage in C2FS is implemented with the help of a coordination service. Three important reasons led us to select this approach instead of, for example, a NoSQL database [Mon, LFKA11] or some custom service (as in other file systems). First, coordination services offer *consistent* storage with enough capacity for this kind of data,[2] and thus can be used as consistency anchors for the cloud storage services. Second, state of the art coordination services [BACF08, Bur06, HKJR10] implement complex replication protocols to ensure *fault tolerance* for the metadata storage. Finally, coordination services support operations with *synchronization* power that can be used to implement fundamental file system functionalities, such as locking.

File data is maintained both in the cloud-of-clouds storage and locally in a cache at the client machine. This strategy is interesting in terms of performance, costs and availability. Since cloud accesses usually entail large latencies, C2FS attempts to keep in the user machine a copy of the accessed files. Therefore, if the file is not modified by another client, subsequent reads do not need to fetch the data from the clouds. As a side effect, there are cost savings as there is no need to pay for the download of the file. On the other hand, we follow the approach of writing everything to the cloud (enforcing *consistency-on-close* semantics [HKM+88]), as most providers let clients upload files for free to create incentive for storing data in the cloud. Consequently, no completed update is lost in case of a local failure.

In our current implementation, the cloud-of-clouds storage is based on the DepSky algorithm, providing a single-writer multiple-reader register abstraction, supporting eventually consistent storage [BCQ+11]. Our coordination service runs on computing clouds, relying on an extended version of DepSpace, which offers a dependable and secure tuple space datastore [BACF08]. Both mechanisms are Byzantine fault-tolerant [CL02], and DepSpace is implemented using the TClouds State Machine Replication component,[3] BFT-SMaRt. The C2FS Agent implements the core algorithms of C2FS, relying on the trusted cloud-of-clouds storage and the coordination service, as detailed in §7.4.3.

[2] Being main-memory databases, their maximum storage capacity is bounded by the amount of RAM of the servers they are deployed.

[3] In fact, during the 2nd year of TClouds we updated the old codebase of DepSpace (described in [BACF08]) for working with BFT-SMaRt and its durability layer (Chapter 3), as described in §3.4.

7.4.3 C2FS Agent

Basic Services

The design of the C2FS Agent is based on the use of three (local) services that abstract the access to the coordination service and the cloud-of-clouds storage:

Storage service. The storage service provides an interface to save and retrieve variable-sized objects from the cloud-of-clouds storage. Since cloud providers are located over the internet, C2FS overall performance is heavily affected by the latency of remote data accesses and updates. To address this problem, we opted for reading and writing whole files as objects in the cloud, instead of splitting them in blocks and accessing block by block. This allows most of the client files (if not all) to be stored locally, and makes the design of C2FS simpler and more efficient for small-to-medium sized files.

To achieve adequate performance, we rely on two levels of caches, whose organization has to be managed with care in order to avoid impairing consistency. First, all files read and written are copied locally, making the local disk a large and long term cache. More specifically, the disk is seen as an LRU file cache with GBs of space, whose content is validated in the coordination service before being returned, to ensure that the most recent version of the file is used. Second, a main memory LRU cache (hundreds of MBs) is employed for holding open files. This is aligned with our consistency-on-close semantics, since, when the file is closed, all updated metadata and data kept in memory are flushed to the local disk and the clouds.

Level	*Location*	*Latency*	*Fault tolerance*	*Sys call*
0	main memory	microsec	none	write
1	local disk	millisec	crash	fsync
2	cloud	seconds	local disk	-
3	cloud-of-clouds	seconds	f clouds	close

Table 7.1: C2FS durability levels and the corresponding data location, write latency, fault tolerance and example system calls.

The actual data transfers between the various storage locations (memory, disk, clouds) are defined by the durability levels required by each kind of system call. Table 7.1 shows examples of POSIX calls that cause data to be stored at different levels, together with their location, storage latency and provided fault tolerance.

For instance, a write in an open file causes the data to be saved in the memory cache, which gives no durability guarantees (Level 0). Calling `fsync` flushes the data (if modified) to the local disk,[4] achieving the standard durability of local file systems, i.e. against process or system crashes (Level 1). In cloud-backed storage systems, when a file is closed, the data is written to the cloud. Since these systems (such as Dropbox or S3FS) are backed by a single cloud provider, they can survive a local disk failure but not a cloud failure (Level 2). However, in C2FS, the data is written to a set of clouds, such that failure of up to f providers is tolerated (Level 3).

Metadata service. The metadata service resorts to the coordination service to store file and directory metadata, together with information required for enforcing access control. In particular, it ensures that each file system object is represented in the coordination service by a metadata tuple containing: the object name, the type (file, directory or link), its parent object (in the hierarchical file namespace), the object metadata (size, date of creation, owner, ACLs,

[4] Alternatively, a file is flushed to disk on each write if the file is opened in synchronous mode.

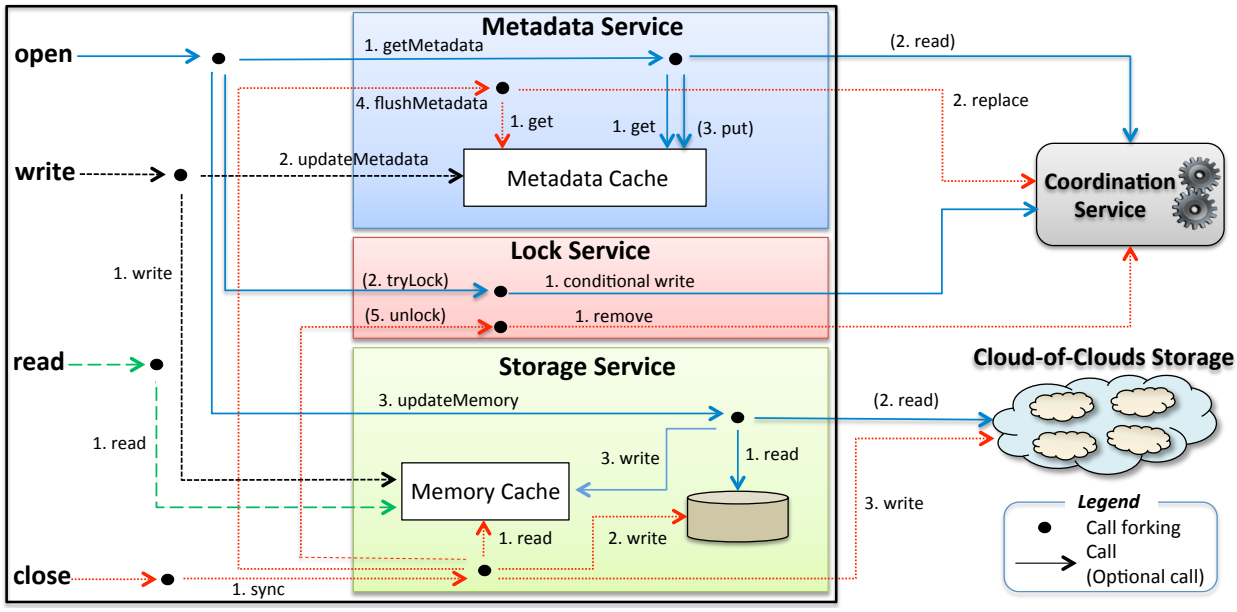


Figure 7.4: Common file system operations in C2FS. The following conventions are used: 1) at each call forking (the dots between arrows), the numbers indicate the order of execution of the operations; 2) operations between brackets are optional; 3) each file system operation (e.g., open/close) has a different line pattern.

etc.), an opaque identifier referencing the file in the storage service (and, consequently, in the cloud-of-clouds storage) and the cryptographic hash of the contents of the current version of the file. These two last fields represent the *id* and hash stored in the constancy anchor (see §7.3). Metadata tuples are accessed through a set of operations offered by the local metadata service, which are then translated into different read, write and replace calls to the coordination service.

To deal with bursts of metadata accesses (e.g., opening a file with the `vim` editor can cause more than five `stat` calls), a small short-lived main memory cache (up to few MBs for tens of millisenconds) is utilized to serve metadata requests. The objective of this cache is to reuse the data fetched from the coordination service for at least the amount of time spent to obtain it from the network.

Locking service. As in most consistent file systems, we use *locks* to avoid write-write conflicts. The locking service implements the following protocol with DepSpace: when a file needs to be locked (e.g., open for writing), the service tries to insert a lock tuple associating the user with the file. Since the insert action is atomic, this ensures that the operation only succeeds if no other lock tuple exists for this file. Therefore, if the tuple is inserted, the client obtains the lock and can use the file exclusively, otherwise the lock fails. Unlocking the file is as simple as deleting the lock tuple from the coordination service.

In order to address faults in clients holding locks for some files, all lock tuples are created with a lease that causes their automatic removal when a certain interval expires. Consequently, the lock tuple needs to be renewed periodically, if a client wants to keep a certain lock.

File Operations

Figure 7.4 illustrates the execution of C2FS when serving the four main file system calls, open, write, read and close. To implement these operations, the C2FS Agent intercepts the system calls issued by the operating system and invokes the procedures provided by the storage, meta-

data and locking services.

Opening a file. The tension between provisioning strong consistency and suffering high latency in cloud access led us to provide *consistency-on-close semantics* [HKM+88] and synchronize files only in the open and close operations. Moreover, given our aim of having most client files (if not all) locally stored, we opted for reading and writing whole files from the cloud. With this in mind, the open operation comprises three main steps: (i) read the file metadata, (ii) optionally create a lock if the file is opened for writing, and (iii) read the file data to the local cache. Notice that these steps correspond to an implementation of the READ algorithm of Figure 7.2, with an extra step to ensure exclusive access to the file for writing.

Reading the metadata entails fetching the file metadata from the coordination service, if it is not available in the metadata cache, and then make an update to this cache. Locking the file is necessary to avoid write-write conflicts, and if it fails, an error is returned. Reading the file data either uses the copy in the local cache (memory or disk) or requires that a copy is made from the cloud. The local data version (if available) is checked to find out if it corresponds to the one in the metadata service. In the negative case, the new version is collected from the cloud-of-clouds and copied to the local storage. If there is no space for the file in main memory (e.g., there are too many open files), the data of the least recently used file is first pushed to disk (as a cache extension) to release space.

Write and read. These two operations only need to interact with the local storage. Writing to a file requires updating the memory-cached file and the associated metadata cache entry (e.g., the size and the last-modified timestamp). Reading just causes the data to be fetched from the main memory cache (as it was copied there when the file was opened).

Closing a file. Closing a file involves the synchronization of cached data and metadata with the coordination service and the cloud-of-clouds storage. First, the updated file data is copied to the local disk and to the cloud-of-clouds. Then, if the cached metadata was modified, it is pushed to the coordination service. Lastly, the file is unlocked if it was originally opened for writing. Notice that these steps correspond to the WRITE algorithm of Figure 7.2.

As expected, if the file was not modified since opened or was opened in read-only mode, no synchronization is required. From the point of view of consistency and durability, a write to the file is complete only when the file is closed, respecting the consistency-on-close semantics.

Other operations. C2FS supports the other file system calls defined in the POSIX standard. There are operations like *link*, *fstat* and *rename* that only require access to the metadata service, while others like *sync* and *fsync* are used to copy the contents of the main memory cache to disk. We restrain from giving more details about these operations due to space constraints.

Garbage Collection

During normal operation, C2FS saves new versions of the file data without deleting the previous ones, and files removed by the user are just marked as deleted in the associated metadata. These two features support the recovery of a history of the files, which is useful for some applications. However, in general this can increase the cost of running the system, and therefore, C2FS includes a flexible garbage collector to enable various policies for reclaiming space.

Garbage collection runs in isolation at each C2FS Agent, and the decision about reclaiming space is based on the preferences (and budget) of an individual user. By default, its activation is guided by two parameters defined upon the mounting of the file system: *number of written bytes* W and *number of versions to keep* V. Every time a C2FS agent writes more than W bytes, it starts the garbage collector as a separated thread that runs in parallel with the rest of the system (other activation policies are possible). This thread fetches the list of files owned by this user

and reads the associated metadata from the coordination service. Next, it issues commands to delete old file data versions from the cloud-of-clouds storage, such that only the last V versions are kept. Additionally, it also eliminates the data versions of the files removed by the user. Later on, the corresponding metadata entries are also erased from the coordination service.

7.4.4 Security Model

The security of a shared cloud-of-clouds storage system is a tricky issue, as the system is constrained by the access control capabilities of the backend clouds. A straw-man implementation would allow all clients to use the same account and privileges on the cloud services, but this has two drawbacks. First, any client would be able to modify or delete all files, making the system vulnerable to malicious users. Second, a single account would be charged for all clients, preventing the pay-per-ownership model.

C2FS implements the enhanced POSIX's ACL model [Gru03], instead of the classical Unix modes (based on *owner, group, others*). The owner O of a file can give access permissions to another user U through the `setfacl` command, passing as parameters the client identifier of U, the permissions and the file name. The `getfacl` command returns the permissions of a file.

As a user has separate accounts in the various cloud providers, and since each probably has a different identifier, C2FS needs to associate with every client identifier a list of cloud canonical identifiers. This association is kept in a tuple in the coordination service, and is loaded when the client mounts the file system for the first time. When the C2FS Agent intercepts a `setfacl` request from a client O to set permissions on a file for a user U, the following steps are executed: (i) O uses the two lists of cloud canonical identifiers (of O and U) to update the ACLs of the objects that store the file data in the clouds with the new permissions; and then, (ii) O also updates the ACL associated with the metadata tuple of the file in the coordination service to reflect the new permissions (see [BACF08]).

Notice that we do not trust the C2FS Agent to implement the access control verification, since it can be compromised by a malicious user. Instead, we rely on the access control enforcement of the coordination service and the cloud providers. However, we do not trust individual providers absolutely: even if up to f_s cloud providers misbehave and give access to a file to some unauthorized user, it would be impossible to retrieve the contents due to the secret sharing and encryption layer implemented by the cloud-of-clouds storage (i.e., DepSky).

Our design assumes that cloud storage providers support ACLs for containers as well as for objects, as implemented in Amazon S3 [S3A] and Google Storage [GOO], and specified in standards such as CDMI [SNI12]. Although some of the providers we currently use do not support ACLs, we expect this feature will be available in the near future.

7.4.5 Private Name Spaces

One of the goals of C2FS is to scale in terms of users and files. However, the use of a coordination service could potentially create a scalability bottleneck, as this kind of service normally maintains all data in main memory (e.g., [BACF08, Bur06, HKJR10]) and requires a distributed agreement to update the state of the replicas in a consistent way. To address this problem, we take advantage of the observation that, although file sharing is an important feature of cloud-backed storage systems, the majority of the files are not shared [DMM+12, LPGM08]. Looking at the C2FS design, all files and directories that are not shared (and thus not visible to other

users), do not require specific tuples in the coordination service, and instead can have their metadata grouped in a single object saved in the cloud-of-clouds storage.

This object is represented with a *Private Name Space* (PNS) abstraction. A PNS is a local object kept by the metadata service of the C2FS agent, containing the metadata of all private files of a user. Each PNS has an associated PNS tuple in the coordination service, which has the user name and a reference to an file in the clouds-of-clouds storage. The file keeps a copy of the serialized metadata of all private files of the user.

Working with non-shared files is slightly different from what was shown in Figure 7.4. When mounting the file system, the agent fetches the user PNS tuple from the coordination service and the metadata from the clouds-of-clouds storage, locking the PNS to avoid inconsistencies caused by two clients logged as the same user. When opening a file, the user gets the metadata locally as if it was in cache (since the file is not shared), and if needed fetches the data from the cloud-of-clouds storage (as in the normal case). On close, if the file was modified, both the data and the metadata are updated in the cloud-of-clouds storage. The close operation completes when both updates finish.

Notice that a file can be removed (or added) from the PNS at any moment if its permissions change, causing the creation (or removal) of a corresponding metadata tuple in the coordination service.

With PNSs, the amount of storage used in the coordination service is proportional to the percentage of shared files in the system. For example, in a setup with 1M files where only 5% of them are shared (e.g., the engineering trace of [LPGM08]): (i) Without PNSs, it would be necessary 1M tuples of around 1KB, for a total size of 1GB of storage (the approximate size of a metadata tuple is 1KB, assuming 100 byte file names); (ii) With PNSs, only 50 thousand tuples plus one PNS tuple per user would be used, requiring a little over 50MB of storage. Even more importantly, by resorting to PNSs, it is possible to reduce substantially the number of accesses to the coordination service, allowing more users and files to be served.

7.5 C2FS Implementation

C2FS is implemented as a file system mounted in user space, using the FUSE-J wrapper to connect the C2FS agent to the FUSE library [FUS]. The C2FS agent was developed in Java mainly because our building blocks – DepSky and DepSpace – were based on Java. Moreover, the high-latencies of cloud accesses make the overhead of using a Java-based file system comparatively negligible.

Building blocks. Our C2FS prototype uses an updated version of DepSpace based on BFT-SMaRt [BSA], a Java state-machine replication library implementing a protocol similar to PBFT [CL02]. We extended DepSpace with some new operations (`replace`, `rdAll`), timed tuples and triggers. Timed tuples have an expiration time, and are fundamental for tolerating faults in the locking protocol. Triggers allow a single tuple update to cause several updates in the tuple space, which are quite useful for implementing file system operations such as `rename`.

We also extended DepSky to support a new operation `readWithHash(du,h)`, which instead of reading the last version of a data unit, reads the version with a given hash h, if available. The hashes of all versions of the file are stored in the DepSky's internal metadata object (not related to C2FS files metadata), stored in the clouds [BCQ+11].

Modes of operation. C2FS supports three modes of operation, based on the required consistency and sharing requirements of the stored data. The first mode, *standard*, is the one described up to this point. The second mode, *non-blocking*, is a weaker version of C2FS in which closing

a file does not block until the file data is on the clouds, but only until it is written locally and enqueued to be sent to the clouds in background. In this model, the file metadata is updated and the its lock released only after the file contents are updated to the clouds, and not on the close call. Naturally, this model leads to a significant performance improvement at cost of a reduction of the durability and consistency guarantees. Finally, the *non-sharing* mode is interesting for users that do not need to share files, and represents a design similar to S3QL [S3Qa], but using a cloud-of-clouds instead of a single storage service. This version does not require the use of the coordination service, and all metadata is stored on private name spaces.

Codebase. Overall, the C2FS implementation requires 17168 lines of code, being 3227 in DepSky, 8010 in DepSpace and the remaining in the C2FS Agent.

7.6 Evaluation

In this section we evaluate the different modes of operation of C2FS and compare them with other cloud-backed file systems. Our main objective is to understand how our system behave with a set of representative workloads and shed light on the costs of our design.

7.6.1 Setup & Methodology

Our setup considers a set of clients running on a cluster of Linux 2.6 machines with two quad-core 2.27 GHz Intel Xeon E5520, 32 GB of RAM memory, and a 146 GB 15000 RPM SCSI hard disk.

We consider a setup with $n_s = 4$ CSSPs, tolerating a single faulty storage cloud. The *storage clouds* used were Amazon S3 (US), Google Storage (US), Rackspace Cloud Files (UK) and Windows Azure (UK). In the same way, we consider three deployments of the coordination service in our evaluation, all of them with $n_c = 4$ replicas (tolerating a single Byzantine fault). First, we set up a cloud-of-clouds (*CoC*) composed of four *computing clouds* – EC2 (Ireland), Rackspace (UK), Windows Azure (Europe) and Elastichosts (UK) – with one coordination service replica on each of them. The second scenario (*EC2*) represents the case in which the coordination service runs in a single *remote* trusted provider, in our case, Amazon EC2 (Ireland). Our third scenario considers all coordination service replicas running *locally* in the same cluster as the clients, representing the case in which the organization supports its users collaboration. In all cases, the VM instances used are EC2' M1 Large [EC2a] (or similar).

We selected a set of benchmarks following some recent recommendations [TBZS11,TJWZ08], all of them based on the *Filebench* benchmarking tool [Fil]. Moreover, we created two new benchmarks for simulating some behaviors of interest for cloud-backed file systems. In all experiments we report averages and standard deviations calculated from at least 100 measurements, discarding 5% of the outliers.

We compare C2FS in different modes of operation, namely: C2FS, NB-C2FS (non-blocking) and NS-CSFS (non-sharing). In all cases we configure the metadata cache expiration time to 500 ms and do not use private name spaces. In §7.6.4 we study variations of these parameters. Moreover, we compare these modes with S3QL [S3Qa] and S3FS [S3F]. Finally, we use a FUSE-J-based local file system (LocalFS) implemented in Java as a baseline. We use such system to ensure an apples-to-apples comparison, since a native file system presents much better performance than a user-space file system based on FUSE-J.

7.6.2 Micro-benchmarks

To better understand the cost of running C2FS, we need to understand the costs of accessing the cloud-of-clouds storage (i.e., DepSky register abstraction) and the coordination service (i.e., DepSpace replicated database).

Cloud-of-clouds storage. Figure 7.5 shows the read and write latencies of DepSky and Amazon S3 considering 8 different file sizes, ranging from 1KB to 16MB.

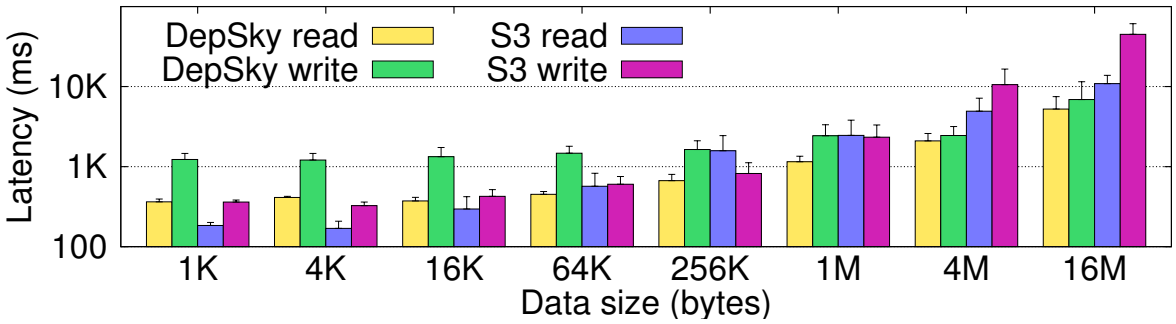


Figure 7.5: Cloud storage latency (milliseconds, log scale) for reading and writing different data sizes using DepSky and S3.

The figure shows that small to medium files (up to around 4MB) can be read by DepSky in less than 2 seconds, and 16MB files are read in less than 6 seconds. Writing files, on the other hand, is roughly $3\times$ slower, with the exception of 16MB files, where a write is 91% slower than a read. When comparing the results of cloud-of-clouds storage with Amazon S3, it can be seen that the latter presents better latency for small files (smaller than 64KB for reads and 1MB for writes). However, for bigger files, a replicated solution is significantly better. Considering reads, this is in accordance with [BCQ+11], since we fetch half of the file[5] from the two fastest clouds on every operation, we get better tolerance to the unpredictable latency of cloud services. In the case of writes, our results show that S3 is substantially less efficient when dealing with 4MB and 16MB files. DepSky avoids this problem by storing only half of the file (2MB and 8MB, respectively) in each cloud, which they can handle proportionally faster.

Coordination service. Table 7.2 shows the latency of some representative metadata service operations that directly translate to coordination service invocations when the metadata cache is not used (as in this experiment).

MS Operation	Local	EC2	CoC
`getMetadata`	2.21 ± 0.85	79.85 ± 0.85	72.81 ± 1.48
`getDir`	3.26 ± 0.45	84.56 ± 1.51	94.03 ± 0.81
`update`	5.13 ± 1.42	86.32 ± 0.83	96.44 ± 1.11
`put`	5.82 ± 0.88	87.15 ± 2.21	96.96 ± 1.36
`delete`	3.31 ± 0.67	84.52 ± 0.91	93.95 ± 0.68

Table 7.2: Latency (ms) of some metadata service operations (no cache) for different setups of DepSpace with (tuples of 1KB).

The table shows the latency difference between running the coordination service inside the same network as the clients (Local) and in a remote location (EC2 and CoC). Overall, locally the operations require from 2-6 ms. When deployed in a remote single cloud (in this

[5]This happens due to the use of erasure codes in DepSky [BCQ+11].

case, EC2 in Ireland), the average latency of 79 ms between our client and the replicas makes the perceived latency increase substantially. Interestingly, the use of cloud-of-clouds does not make the latency of the metadata service operations substantially worse. In fact, the read-only `getMetadata` operation, that returns a single metadata tuple, is faster in the CoC case due to the smaller average latency between the client and the replicas, 65 ms. The `getDir` operation, which is also read-only, is slower for two factors: the search for all matching tuples in the system is much slower and the reply size is much bigger (several tuples) when compared with `getMetadata`. For the other operations, which require the execution of a Byzantine agreement [BACF08], there is also an average increase of 10 ms when compared with EC2. This, together with a 14 ms difference between the client-replicas latency in these two settings, shows that running the Byzantine agreement protocol between the replicas in different clouds takes roughly 24 ms. This makes sense since the inter-clouds latency ranges from 2.5 ms (Azure-EC2) to 16 ms (Azure-Elastichosts), with an average of 10.6 ms, which is substantially smaller than the client-replica latency.

Filebench micro-benchmarks. Table 7.3 shows the results for different Filebench micro-benchmarks considering all setups and modes of operation for C2FS.

Micro-benchmark	#Operations	File size	C2FS			S3FS	NB-C2FS			NS-C2FS	S3QL	LocalFS
			Local	EC2	CoC		Local	EC2	CoC			
sequential read	1	4MB	1	1	1	6	1	1	1	1	1	1
sequential write	1	4MB	1	1	1	2	1	1	1	1	1	1
random 4KB-read	256k	4MB	11	11	11	15	11	11	11	11	11	11
random 4KB-write	256k	4MB	36	34	36	52	35	33	35	35	152	37
create files	200	16KB	156	282	321	596	5	85	95	1	1	1
copy files	100	16KB	430	467	478	444	3	84	94	1	1	1

Table 7.3: Latency (in seconds) of several Filebench micro-benchmarks for three variants of C2FS, S3QL, S3FS and LocalFS.

The considered benchmarks are the following [Fil]: sequential reads, sequential writes, random reads, random writes, create files and copy files. The first four benchmarks are IO-intensive and do not consider open and close operations, while the last two are metadata-intensive. The results of the sequential and random r/w benchmarks show that the behavior of the evaluated file systems is similar, with the exception of S3FS and S3QL. S3FS low performance comes from its lack of main memory cache for opened files [S3F], while S3QL low random write performance is the result of a known issue with FUSE, that makes small chunk writes very slow [S3Qb]. This benchmark performs 4KB-writes, much smaller than the recommended chunk size of S3QL (128KB).

The results for create files and copy files show the difference between running a local or single-user cloud-backed file system (NS-C2FS, S3QL and LocalFS) and a shared and/or blocking cloud-backed file system (C2FS, S3FS and NB-C2FS): three to four orders of magnitude. This is not surprising, given that C2FS, NB-C2FS and S3FS access a remote service for each create, open or close operation. Furthermore, NB-C2FS' latency is dominated by the coordination service operations' latency, while in C2FS the latency is dominated by the read/write operations in the cloud-of-clouds storage. This makes the standard C2FS almost not sensitive to the setup of the coordination service. In the end, these results confirm that running the coordination service in EC2 or in a CoC have little impact in terms of performance, despite being a huge improvement in terms of failure independence.

7.6.3 Application-based Benchmarks

In this section we present some application-based benchmarks illustrating different uses of C2FS.

Personal storage service. A representative workload of C2FS is one that corresponds to its use as a personal cloud storage service [DMM+12] in which desktop application files (e.g., xlsx, docx, pptx, odt, pdf) are stored and shared. We designed a new benchmark to simulate opening, saving and closing actions on a text document (odt file) in the OpenOffice application suite. The benchmark follows the behavior observed in traces of the real system. As expected, OpenOffice works similarly as other modern desktop applications [HDV+11]. However, the files under C2FS are just copied to a temporary directory on the local file system where they are manipulated as described in [HDV+11]. Nonetheless, as can be seen in the benchmark definition (Figure 7.6), these actions (specially save) still impose a lot of work on the cloud-backed file system.

Open Action: 1 open(f,rw), 2 read(f), 3-5 open-write-close(lf1), 6-8 open-read-close(f), 9-11 open-read-close(lf1)

Save Action: 1-3 open-read-close(f), 4 close(f), 5-7 open-read-close(lf1), 8 delete(lf1), 9-11 open-write-close(lf2), 12-14 open-read-close(lf2), 15 truncate(f,0), 16-18 open-write-close(f), 19-21 open-fsync-close(f), 22-24 open-read-close(f), 25 open(f,rw)

Close Action: 1 close(f), 2-4 open-read-close(lf2), 5 delete(lf2)

Figure 7.6: File system operations invoked in the personal storage service benchmark, simulating an OpenOffice text document open, save and close actions (f is the odt file and lf is a lock file).

Table 7.4 shows the average latency of each of the three actions of our benchmark for LocalFS, S3QL, C2FS and S3FS, considering a file of 1.2MB, which corresponds to the average file size observed in 2004 (189KB) scaled-up 15% per year to reach the expected value for 2013 [ABDL07].

Action	LocalFS	S3QL	NS-C2FS	NB-C2FS (CoC)	C2FS (CoC)	S3FS
Open	20.5 ± 1.6	5.7 ± 0.6	20.5 ± 6.1	779.6 ± 282.7	4812.8 ± 1655.3	4485 ± 143.2
Save	54.8 ± 9.8	78.6 ± 8.6	100.7 ± 21.1	1169 ± 875.8	16859.3 ± 6556.9	13677.1 ± 1730.7
Close	0.6 ± 0.5	1.02 ± 0.1	1.5 ± 0.5	190.1 ± 44.4	260.3 ± 3.55	800.3 ± 13.6
Open (L)	19 ± 0.8	3.06 ± 0.7	17.5 ± 0.7	19.6 ± 16.6	222.9 ± 7.6	817.5 ± 31.3
Save (L)	50.6 ± 1.2	56.2 ± 8.77	87.4 ± 21	131.2 ± 71.8	7703.5 ± 2619	9917.6 ± 1753.3
Close (L)	0.06 ± 0.2	0.05 ± 0.02	0.07 ± 0.26	0.15 ± 0.36	0.25 ± 0.44	271.7 ± 12.1

Table 7.4: Latency and standard deviation (ms) of a personal storage service actions in a file of 1.2MB. The (L) variants maintain lock files in the local file system. *C2FS and NB-C2FS uses a CoC coordination service*.

The results show that NS-C2FS presents the best performance among the ones we evaluated, having a performance very similar to a local file system, where a save takes around 100 ms. Comparing it with S3QL and S3FS, we can see that using a cloud-of-clouds storage as a backend for a file system can be similar to a local file system, if the correct design decisions are taken.

The NB-C2FS requires substantially more time for each phase due to the number of accesses to the coordination service, specially to deal with the lock files used in this workload. Nonetheless, saving a file in this system takes around 1.2 seconds, which is acceptable from the usability point of view.

A even slower behavior is observed in the (blocking) C2FS, where when a lock file is created, the system blocks waiting for this small file to be pushed to the clouds. This makes

the save operation extremely slow. Interestingly, the latency of save in C2FS is close to S3FS, which also writes synchronously to the cloud, and S3QL which, as already mentioned, performs poorly with small file writes.

We observed that most of the latency of these actions comes from the manipulation of lock files. However, these files do not need to be stored in the C2FS partition, since we already avoid write-write conflicts. We modified the benchmark to represent an application that writes lock files locally (e.g., in `/tmp`), just to avoid conflicts between applications in the same machine. The three last rows of the table present these results, which show that removing such lock files makes the cloud-backed system much more responsive. The takeaway here is that the usability of C2FS could be substantially improved if applications take into consideration the limitations of accessing remote services.

Sharing files. Personal cloud storage services are often used for sharing files in a controlled and convenient way [DMM+12]. We designed an experiment for comparing the time it takes for a shared file written by a client to be available for reading by another client, using C2FS and non-blocking C2FS, both using a CoC coordination service. We did the same experiment considering a Dropbox shared folder (creating random files to avoid deduplication). We acknowledge that the Dropbox design [DMM+12] is quite different from C2FS, but we think it is illustrative to show how a cloud-of-clouds personal storage service might compare with a popular system.

The experiment considers two clients A and B deployed in our cluster. We measure the elapsed time between the instant client A closes a variable-size file that it wrote to a shared folder and the instant it receives an UDP ACK from client B informing the file was available. Clients A and B are Java programs running in the same LAN, with a ping latency of around 0.2 milliseconds, which is negligible considering the latencies of reading and writing. Table 7.5 shows the results of this experiment for different file sizes.

File Size	C2FS	NB-C2FS	Dropbox
256 KB	1172 ± 82.4	6408 ± 735	9895 ± 10667
1 MB	1679 ± 170.6	8532 ± 746	12070 ± 7783
4 MB	3915 ± 408.2	20016 ± 913	20511 ± 1797
16 MB	12281 ± 1081	68211 ± 6127	83535 ± 3526

Table 7.5: Sharing file latency (ms) for C2FS (in the CoC) and Dropbox for different file sizes.

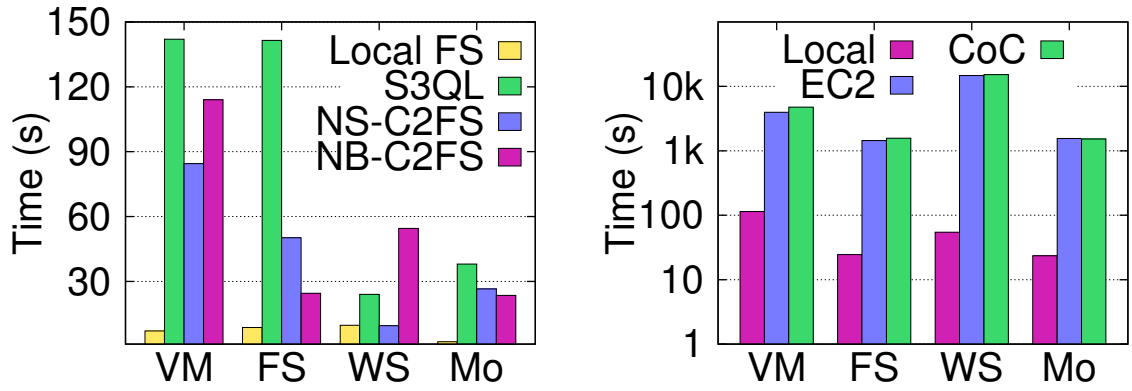
The results show that the latency of sharing in C2FS is significantly better than what people experience in current personal storage services. These results do not consider the benefits of deduplication, which C2FS does not support. However, if a user encrypts its critical files locally before storing them in Dropbox, the effectiveness of deduplication will be decreased significantly.

Table 7.5 also shows that the latency of the standard C2FS is much smaller than the non-blocking version (C2FS-NB). This is explained by the fact that C2FS waits for the file write to complete before returning to the application, making the benchmark measure only the delay of reading the file. This illustrates the benefits of C2FS: when A completes its file closing, it knows the data is available to any other client the file is shared with.

Server workloads in C2FS. This section considers the scenario in which critical servers maintain their files in C2FS to benefit from its durability (e.g., for automatic backup and disaster recovery). For this scenario, we consider a client running a service that imposes some file system workload. This means there is a set of files being updated by a single client, thus we con-

sider NS-C2FS and NB-C2FS, which update the cloud in background. The following Filebench application-emulation workloads [Fil] were considered: *VarMail* (VM; similar to Postmark, but with multiple threads [TJWZ08]), *FileServer* (FS; similar to SPECsfs), *WebServer* (WS; mostly read workload) and *Mongo* (Mo; file system activity of the MongoDB NoSQL database [Mon], where 16KB files are appended, read and eliminated).

Figure 7.7 shows the results of executing these benchmarks with different setups of NB-C2FS and NS-C2FS, together with S3QL and LocalFS.



(a) C2FS vs S3QL vs Local FS. (b) NB-C2FS (Local, EC2, CoC).

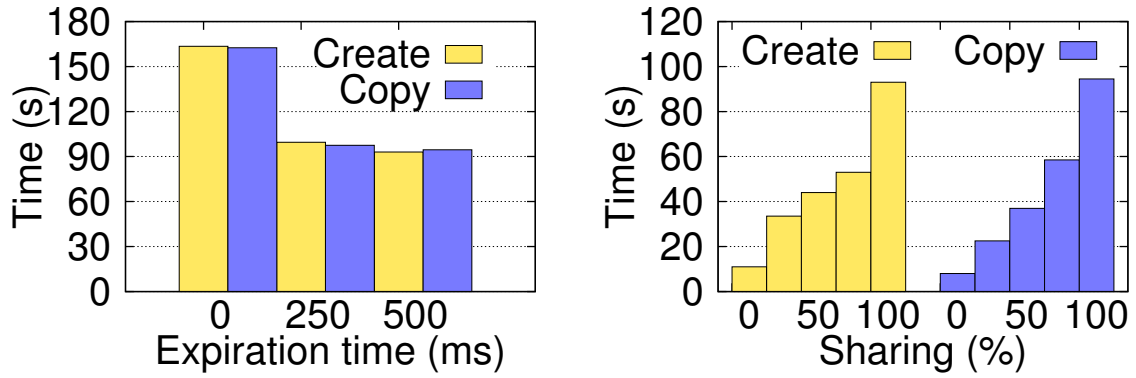
Figure 7.7: Execution time of Filebench application benchmarks for: (a) LocalFS, S3QL, NS-C2FS and NB-C2FS (Local); and (b) NB-C2FS in different storage setups.

In general, these benchmarks are not favorable to S3QL, since all of them contain small data writes. Moreover, one can see that NB-C2FS and NS-C2FS present a latency much higher than LocalFS. The only exception is for WebServer, a mostly-read benchmark, with a relatively small working set. For the other benchmarks, the cost of managing metadata plays an important role. In particular, the NB-C2FS coordination service accesses (even in a local network) make it 3-10× slower than LocalFS. Interestingly, NB-C2FS is faster than NS-C2FS in the FileServer and Mongo benchmarks, even with the lack of the coordination service in the latter. This happens because these benchmarks use a lot of files, that need to have their metadata updated in each write. This implies the serialization and transmission of the whole metadata collection (a PNS) to the clouds, which is done too many times. This could be alleviated by dividing the metadata collection in multiple, smaller, PNSs. We note these application-benchmarks were not modified for a cloud-backed file system, and in this sense our results are promising: it appears to be possible to modify applications like these to make use of cloud-backed file systems more efficiently.

Figure 7.7(b) shows that running NB-C2FS in remote service increases the benchmarks latency 2-3 orders of magnitude. This is explained by the difference in the latency of accessing the remote coordination service, which indicates that such workloads might be too intensive for a NB-C2FS with a remote coordination service. The tests confirm the trend observed in Tables 7.2 and 7.3, where deploying the coordination service in EC2 or in the CoC makes almost no difference in the system performance.

7.6.4 Varying C2FS Parameters

Figure 7.8 shows some results for two metadata-intensive micro-benchmarks (copy files and create files) considering some variations of two C2FS parameters.



(a) Metadata cache.

(b) Private Name Spaces.

Figure 7.8: Effect of metadata cache expiration time (ms) and PNSs with different file sharing percentages in two metadata intensive micro-benchmarks.

As described in §7.4, we implemented a short-lived metadata cache to deal with bursts of metadata access operations (e.g., `stat`). All experiments consider an expiration time of 500 ms for this cache. Figure 7.8(a) shows how changing this value affects the performance of the system. The results clearly indicate that not using such metadata cache (expiration time equals zero) severely degrades the system performance. However, beyond some point, increasing it does not bring much benefit either.

Figure 7.8(b) shows the latency of the same benchmarks considering the use of PNS (Personal Name Spaces - see §7.4.5) with different percentages of files shared between more than one user. Recall that all previous results consider full-sharing, without using PNS. As expected, the results show that as the number of private files increases, the performance of the system improves. For instance, when only 25% of the files are shared – more than what was observed in the most recent study we are aware of [LPGM08] – the latency of the benchmarks decreases by a factor of roughly 2.5 (create files) and 3.5 (copy files).

7.6.5 Financial Evaluation

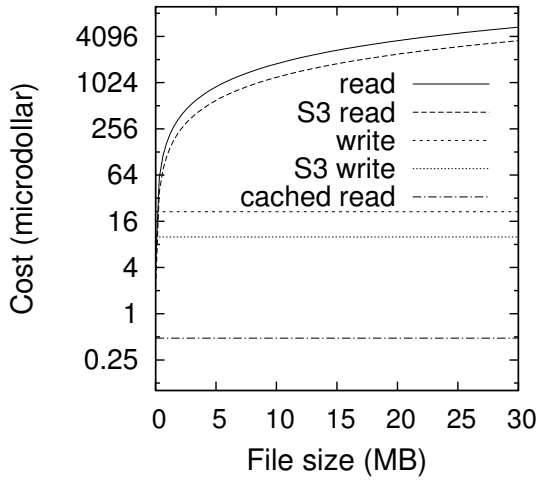
In this section we analyze the costs associated with the deployment and operation of C2FS in public clouds. All calculations are based on the providers costs for storage, computing (pay-as-you-go hourly), outbound data transfer and invoked operations [EC2b, EH-, GS-, RS-a, RS-b, WA-, S3-], ignoring some providers limited free or promotional offers.

Figure 7.9 shows the costs associated with operating and using C2FS. The operating costs of C2FS comprise mainly running the coordination service, which, for our setup with $f_c = 1$, requires four VMs deployed either in a single cloud or in a CoC. Figure 7.9(a) considers the two instance sizes (as defined in Amazon EC2) and the price of renting four of them in Amazon EC2 or in the CoC (one VM of similar size for each provider), together with the expected storage (in number of 1KB-metadata tuples) and processing capacity (in read/write 1KB-kops/s) of such DepSpace setup.[6] As can be seen in the figure, a setup with four Large instances would cost less than $1200 per month while such setup in EC2 would cost $749. This difference of $451 can be seen as the *operating cost* of tolerating one provider failure in our C2FS setup, and this overhead comes from the fact that Rackspace and Elastichosts charge almost 100% more than EC2 and Azure for the same expected VM instance. Moreover, such costs can be factored

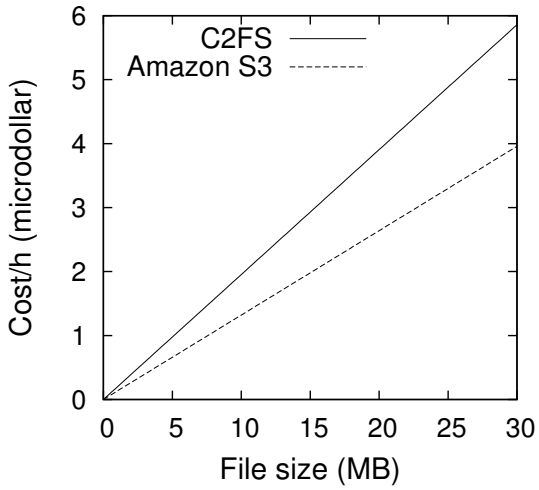
[6] Measured in our local cluster, with VMs with similar capacity.

VM Instance	EC2	CoC	Storage	Processing
Large	$24.96	$39.60	7M files	10 kops/s
Extra Large	$51.84	$77.04	15M files	15 kops/s

(a) Operating costs (day) and expected DepSpace capacity.



(b) Cost per operation (log scale).



(c) Cost per file per hour.

Figure 7.9: The operating and usage costs of C2FS. The costs include outbound traffic generated by the coordination service protocol for metadata tuples of 1KB.

among the users of the system, e.g., for one dollar per month, 2300 users can have a C2FS setup with Extra Large replicas for the coordination service. Finally, this explicit cost of operating the system can be eliminated if the organization using C2FS hosts the coordination service in its infrastructure (our Local setup).

Besides the operating costs, each C2FS user has to pay for its usage (operation and storage) of the file system. Figure 7.9(b) presents the cost of reading a file (open for read, read whole file and close) and writing a file (open for write, write the whole file, close) in C2FS and S3 (e.g., using S3FS), together with the cost of reading a cached file in C2FS. The cost of reading a file is the only one that depends on the size of data, since providers charge around $0.12 per GB of outbound traffic, while inbound traffic is free. Besides that, there is also the cost associated with the `getMetadata`, used for cache validation, which is 11.32 microdollars ($\mu$$11.32). This is the only cost of reading a cached file. The cost of writing is composed by metadata and lock service operations (see Figure 7.4), since inbound traffic is free. Notice that the design of C2FS exploits these two points: unmodified data is read locally and always written to the cloud for maximum durability.

Storage costs in C2FS are charged per version of the file created. Figure 7.9(c) shows the cost/file/hour in C2FS considering the use of erasure codes and preferred quorums [BCQ+11]. When comparing with Amazon S3 alone, the storage costs of C2FS are roughly 50% more.

The storage costs are important because they can be used to define when it is cost-effective to run the garbage collector to delete old files. Running the garbage collector imposes at least the cost of listing the user objects and deleting the ones presenting file versions that are not needed anymore. Currently, two clouds charge $\mu$$1 per list operation, while two others allow these operations for free. All clouds support free deletes. So, running the garbage collector costs $\mu$$2, independently of the number of files to be deleted.

Storing data in the CoC costs $\mu$$0.2 MB/hour (three clouds storing half MB each), con-

sequently, it is economically advantageous to run the garbage collector only when the cost of storing old and unnecessary versions in the next hour is bigger than the cost of running the GC. In our setup, this happens when this unneeded storage reaches 100 MB×hours. This means that after writing 100MB of data in one hour, the system should run the GC, or after keeping 10MB for 10 hours, it is better to run the GC instead of paying for one more hour. Notice that in the second case it makes sense to wait to run the GC, to dilute its cost by reclaiming more storage.

7.7 Related Work

The distributed file systems literature is vast. Here we discuss some of the works we think are relevant to C2FS.

Cloud-of-clouds storage. The use of multiple (unmodified) cloud storage services for data archival was first described in RACS [ALPW10]. The idea is to use RAID-like techniques to store encoded data in several providers to avoid vendor lock-in problems, something already done in the past, but requiring server code in the providers [KAD07a]. DepSky [BCQ+11] integrates such techniques with secret sharing and Byzantine quorum protocols to implement single-writer registers tolerating arbitrary faults of storage providers. ICStore [BCE+12] showed it is also possible to build multi-writer registers with additional communication steps and tolerating only crash faults/unavailability of providers. The main difference between these works and C2FS is the fact they provide a basic storage abstraction (a register), not a complete file system. Moreover, they provide strong consistency only if the underlying clouds provide it, while C2FS uses a consistency anchor (a coordination service) for providing strong consistency independently of the guarantees provided by the storage clouds.

Cloud-backed file systems. S3FS [S3F] and S3QL [S3Qa] are two examples of cloud-backed file systems. Both these systems use unmodified cloud storage services (e.g., Amazon S3) as their backend storage. S3FS employs a blocking strategy in which every update on a file only returns when the file is written to the cloud, while S3QL writes the data locally and later pushes it to the cloud. An interesting design is implemented by BlueSky [VSV12], another cloud-backed file system that can use cloud storage services as a storage backend. BlueSky provides a CIFS/NFS proxy (similar to several commercially available cloud storage gateways) to aggregate writings in log segments that are pushed to the cloud in background, implementing thus a kind of log-structured cloud-backed file system. These systems differ from C2FS in many ways, but mostly regarding their dependency of a single cloud provider and lack of controlled sharing support.

Untrusted cloud providers. There are many recent works providing layers of integrity protection over untrusted storage services. SUNDR [LKMS04] is a file system that ensures fork-consistency over untrusted storage elements. In practice, such file system allows verification of stored files integrity (i.e., if they contain only the updates from authorized clients) even if the storage server misbehaves. Depot [MSL+10] extended these ideas and applied them to cloud storage services. Another recent system, Iris [SvDJO12], provides a file system interface that uses the cloud as storage backend with the help of a proxy, like BlueSky. The downside of these techniques is that they cannot be used in cloud storage services as they require specific code to run in the cloud and do not tolerate unavailability of providers, but rely on collaborating (correct) clients for data recovery in case of data loss/corruption.

Wide-area file systems. Starting with AFS [HKM+88], many file systems were designed for geographically dispersed locations. AFS introduced the idea of copying whole files from the servers to the local cache and making file updates visible only after the file is closed. C2FS

adapts both these classical features for a cloud-backed scenario.

File systems like Oceanstore [KBC+00], Farsite [ABC+02] and WheelFS [SSZ+09] use a small and fixed set of nodes as locking and metadata/index service (usually made consistent using protocols like Paxos [Lam98] and PBFT [CL02]). Similarly, C2FS requires a small amount of computing nodes to run a coordination service and simple extensions would allow C2FS to use multiple coordination services, each one dealing with a subtree of the namespace (improving its scalability) [ABC+02]. However, contrary to these systems, C2FS requires only these "explicit" servers, since the storage cloud services replace the storage nodes.

Parallel file systems. The idea of separating data and metadata storage is used in many parallel file systems (e.g., Hadoop FS [SKRC10], Ceph [WBM+06], PanFS [WUA+08]) for isolating the performance of metadata and data operations. Moreover, such an architecture allows the splitting and distribution of blocks/files through many data nodes for achieving parallel I/O and increasing data transfer bandwidth in tightly coupled clusters. In C2FS, we separate data and metadata not only for performance reasons, but also for flexibility and consistency.

7.8 Conclusions

C2FS is a cloud-backed file system that can be used for backup, disaster recovery and controlled file sharing, without requiring trust on any single cloud provider. A key enabling factor of our design is the reuse of TClouds core sub-systems – DepSky (Resilient Object Storage) and DepSpace (State Machine Replication, more specifically, BFT-SMART) –, achieving high levels of dependability without having to develop complex fault-tolerant protocols for data and metadata storage. This allowed us to focus on the interaction between these systems to design the best possible system given our goals: security, consistency and cost-efficiency.

Chapter 8

Towards Privacy-by-Design Peer-to-Peer Cloud Computing

Chapter Authors:
Leucio Antonio Cutillo (POL).

8.1 Introduction

The *Everything as a Service* paradigm proposed by Cloud computing is changing *de facto* the way Internet users, such as individuals, institutions and companies, deal with data storage and computation. Globally deployed cost-efficient and scalable resources are made available on demand, allowing users to access them via lightweight devices and reliable Internet connection. In recent reports, Comscore pointed out that 9.4 million new smartphones were acquired in EU5[1] in December 2012, and 136 million people now have a smartphone in this area [Com13a]. Moreover, 92% of the world's data has been created in just the last two years, and right now popular Cloud platforms such as YouTube store 72 hours of new videos every minute [Com13b].

Evidence shows that the benefits from apparently unlimited resources come at extremely high security and privacy costs [SK11]. User identity, location and activity information is constantly uploaded and synchronized at Cloud providers' facilities through mobile services, online social networks, search engines and collaborative productivity tools. Such data can be misused both by the provider itself and by attackers taking control of it. Additionally, due to the huge user base, Denial of Service (DoS) attacks reveal to be more effective.

While dependability can be addressed by running several service instances on different Clouds at increased costs, therefore moving to the so-called ***Cloud-of-clouds***, security and privacy vulnerabilities still remain open issues and have a severe impact on user trust in the Cloud [AI12].

Adoption of an appropriate encryption mechanism may appear a viable solution to protect user privacy. Unfortunately, securing outsourced data and computation against untrusted Clouds through encryption is cost-unfeasible [CS10], being outsourcing mechanisms up to several orders of magnitude costlier than their non-outsourced, locally run, alternatives. Moreover, the simple use of encryption to provide data confidentiality and integrity fails to hide sensitive information such as user identity and location, session time and communication traces.

However, even at the presence of fine-grained, cost-effective security and privacy protection tools based on encryption, current Cloud solutions would still suffer from a main orthogonal problem: the intrinsic contrast between their business model and user privacy. As a matter of fact, all current Cloud services are run by companies with a direct interest in increasing their user-base and user demand; service level agreements are often stringent to the user, and

[1] UK, Germany, France, Italy and Spain.

countermeasures against privacy violations are usually taken a-posteriori, once the violation has been detected. Given the promising public Cloud service market size, which is estimated to grow from $129.9 billion in 2013 to $206.6 billion in 2016 [Gar12], Cloud providers are not likely to address this problem in the near future.

In this chapter, we assume the protection of user privacy against the omniscient Cloud provider to be the main objective for Clouds, and we present a sketch for our novel approach to Cloud-of-clouds services that helps to better protect the security of users while allowing for the full scale of operations they are used to from existing Clouds.

The main contributions of this chapter are two: (i) to facilitate confidentiality and privacy by avoiding potential control from any central omniscient entity such as the Cloud provider through a distributed architecture for Cloud-of-clouds, where each Cloud user is a Cloud service provider too; (ii) to leverage on the real life trust relationships among Cloud users to lower the necessity for cooperation enforcement with respect to Cloud service availability. As an additional objective, the protection of the user's privacy against malicious users is also addressed.

The proposed architecture aims at preserving the user's privacy from the outset, and targets ***privacy-by-design***.

This chapter is organized as follows: section 8.2 introduces the main security objectives we expect to meet with our novel approach, which is presented in section 8.3 and detailed in section 8.4; section 8.5 provides a preliminary evaluation of the approach against such objectives, while section 8.6 presents the related work. Finally, section 8.7 concludes this chapter and outlines future work.

8.2 Security objectives

We assume the protection of the user's privacy against the omniscient Cloud service provider to be the main objective for Cloud services.

Privacy. Privacy is a relatively new concept, born and evolving together with the capability of new technologies to share information. Conceived as *"the right to be left alone"* [WB90] during the period of newspapers and photographs growth, privacy now refers to the ability of an individual to control and selectively disclose information about him.

The problem of users' data privacy can be defined as the problem of *usage control* [PS02], which ensures access control together with additional control on the later usage of the data, even once information has already been accessed. Access to the content of user-generated data should only be granted by the user directly, and this access control has to be as fine-grained as specified by the user.

In addition, communication privacy calls for inference techniques aiming at deriving any type of information with regard to: (1) *anonymity*, meaning that users should access resources or services without disclosing their own identities; (2) *unobservability*, i.e. the requirement that no third party should gather any information about the communicating parties and the content of their communication; (3) *unlinkability*, which requires that obtaining two messages, no third party should be able to determine whether both messages were sent by the same sender, or to the same receiver; (4) *untraceability*, which demands that no third party can build a history of actions performed by arbitrary users within the system; in other words, it demands both anonymity and unlinkability.

In summary, the objective of privacy is to hide any information about any user at any time, even to the extent of hiding their participation and activities within the Cloud service in the first place. Moreover, privacy has to be met by default, i.e. all information on all users and their

actions has to be hidden from any other party internal or external to the system, unless explicitly disclosed by the users themselves.

Integrity. In Cloud services, any unauthorized modification or tampering of user-generated information has to be prevented. This encompasses the protection of real identity of users within the Cloud platforms. In this sense, the definition of integrity is extended in comparison with the conventional detection of modification attempts on data. Moreover, problems with integrity of user profiles and their contents may have devastating impact on the objectives put forth with respect to the privacy of Cloud users. Since the creation of profiles in popular Cloud services is easy, protection of real identities is insufficient in today's platforms. In particular, providers offering Cloud services for free are often unable (and perhaps even not interested in) to ensure that a profile is associated to the corresponding individual from the real world.

Availability. The objective of availability for Clouds aims at assuring the robustness of the services in the face of attacks and faults. Due to their exposure as single points of failure, centralized Cloud services are exposed to *denial-of-service* attacks, which directly impact the availability of user's data.

Also distributed services, which are implemented in a decentralized way, possibly via peer-to-peer systems, or which follow other types of service delegation, may be vulnerable to a series of attacks against availability as well. These attacks include *black holes*, aiming at collecting and discarding a huge amount of messages; *selective forwarding*, where some traffic is forwarded to the destination, but the majority is discarded; and *misrouting*, which aims to increase the latency of the system or to collect statistics on the network behavior. In any case, attacks on distributed Cloud systems are more effective in case of *collusion* amongst malicious users or in the presence of Sybil nodes controlled by the attacker, which is not the case for the centralized Cloud providers.

8.3 A new approach

Our system provides Cloud services based on a peer-to-peer architecture. The peer-to-peer architecture meets the privacy concerns by avoiding potential control and misuse of user's data from the omniscient Cloud service provider or attackers taking control of it. Furthermore, cooperation among peers is enforced by leveraging on the real life trust relationships among the user themselves. Each participant is associated to a ***User Identifier*** (UId) and joins the network from multiple devices associated to different ***Node Identifiers*** (NIds). Resources of the participant's devices are available to the participant himself, and to the participant's trusted contacts and contacts-of-contacts with the participant's consent.

8.3.1 System Overview

Our system consists of three main components (Fig. 8.1): a ***Web of Trust*** (WoT), a ***Distributed Hash Table*** (DHT), and a series of ***Trusted Identification Services*** (TISs).

The WoT provides the basic distributed structure used to supply Cloud services, the DHT provides a basic dictionary service to perform lookups, finally each TIS serves the purpose of user authentication.

Web of Trust. The WoT (Fig. 8.2) is a digital mapping of the trust relationships users entertain in their real life, and serves the purpose of Cloud service provisioning. In a user's WoT view,

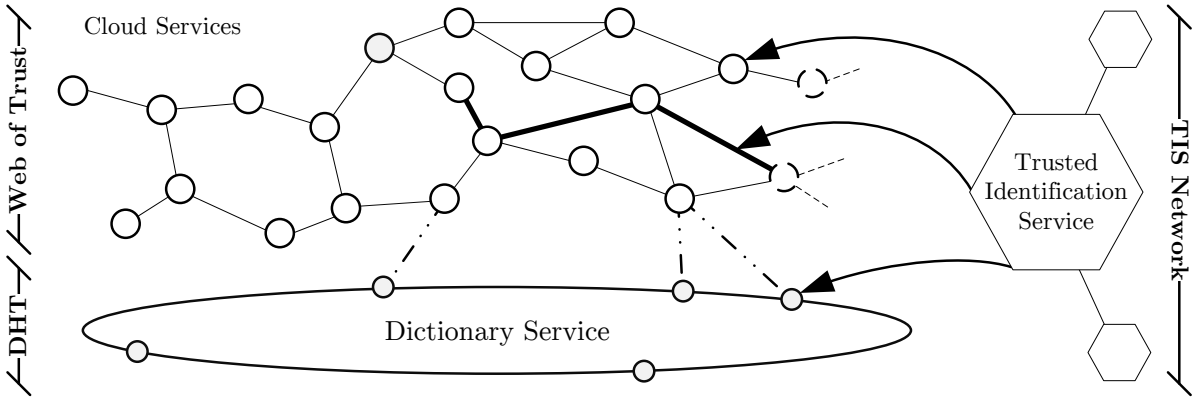


Figure 8.1: Main components of the system: Distributed Hash Table, Web of Trust, and Trusted Identification Service network.

each user's trusted contact acts as a ***Trusted Cloud Service Provider*** (TCSP), and provides the user with storage and computing resouces. Such resources can be allocated both on the TCSP hardware and in that one of its respective TCSPs ones. However, since we don't assume transitivity of trust, a TCSP of a user's TCSP is considered an ***Untrusted Cloud Service Provider*** (UCSP). To preserve both the consumer's privacy and the trust edges in the WoT, UCSP resources are transparently accessed through TCSP only. Finally, in case a TCSP is offline, the set of UCSP resources accessible through that TCSP is still reachable through a set of ***Auxiliary Access Points*** (AAPs), which lead to the TCSP contacts through ***random walks*** on the WoT graph.

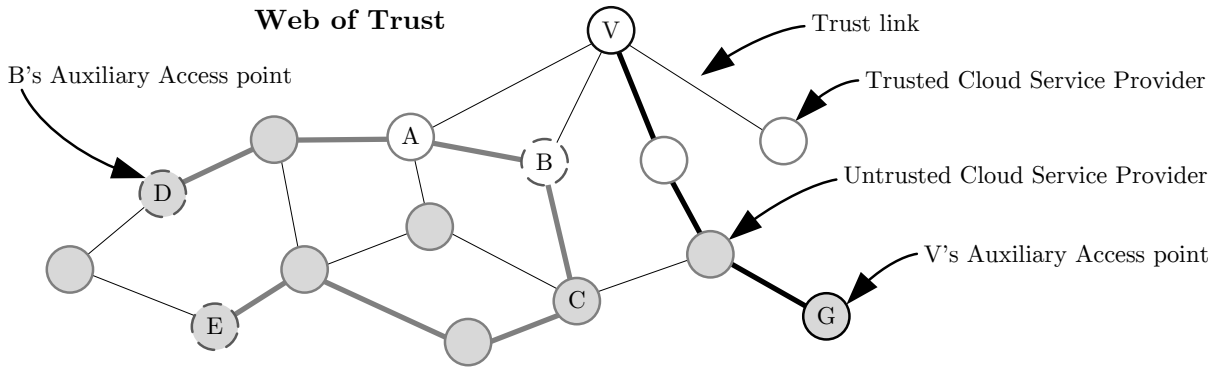


Figure 8.2: The Web of Trust Component: in white, Trusted Cloud Service Providers (trusted contacts) for $\mathcal{V}$; in light gray, Untrusted Cloud Service Providers for $\mathcal{V}$. Node $\mathcal{B}$ is offline, part of the services $\mathcal{B}$ provides to $\mathcal{V}$ are still accessible from $\mathcal{B}$'s Auxiliary Access Points $\mathcal{D}$ and $\mathcal{E}$. Random walks in light gray forward $\mathcal{V}$'s request for $\mathcal{B}$'s services to $\mathcal{B}$'s direct contacts $\mathcal{A}$ and $\mathcal{C}$ without revealing the real requester $\mathcal{V}$'s identity.

DHT. The DHT is implemented by an overlay on top of the internet where peers are arranged thanks to their NId and serves the purpose of TCSP lookup. The DHT is maintained by the users' nodes and provides three distinct lookup services: it returns IP addresses associated to a target NId; it returns a list of AAP for a given UId; it returns the UId associated to a target TCSP identity, such as the user's full name.

Therefore, the DHT allows for building the WoT overlay and addressing the TCSP services.

TIS network. TISs are independent trusted parties serving the purpose of user and device authentication. A TIS provides each user with a certified UId and a set of certified NIds, one for each user device. Any TIS computes identifiers starting from the user's real identity by running the same algorithm.

TISs are offline entities contacted at the first join and do not play any role neither in the communication between users nor in the Cloud service provisioning. Consequently, they do not break the main purpose of decentralization.

8.3.2 Orchestration

A newcomer generates a series of public-private key pairs, contacts a TIS and obtains as an answer his User- and Node- Identifiers, together with certificates associating each identifier with a public key. The newcomer device joins the DHT thanks to the Node Identifier, and the newcomer starts looking for trusted contacts from a series of properties such as the contact name. As an answer, the newcomer receives a set of AAPs for different User Identifiers, and starts retrieving publicly available profile data associated to each identifier through the respective AAPs. Once identified the correct trusted contact, the newcomer sends a contact request to the AAPs which is forwarded along a random walk on the WoT graph. Replies are forwarded back along the same path. A new trusted link is therefore established in the WoT graph. Contact requests contain available devices Node Identifiers and a series of secrets to access their running services. The newcomer queries the P2P system for the IP addresses of each device, and establishes one-hop connections with them. The newcomer then sends to the trusted contact a random walk request, which will create a random walk ending to an AAP for the newcomer. By repeating the abovementioned process, the newcomer adds further real-life trusted contacts and creates a random walk for each of them. The newcomer's trusted contacts act as TCSPs and provide services encompassing Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS).

8.4 Operations

In the following, we sketch the operations implementing the distributed Cloud-of-Clouds, which consist of: (i) account creation, (ii) trusted contact establishment, and (iii) Cloud service access.

Each operation calls for the execution of a series of secure protocols aiming at obtaining credentials, building and keeping the consistency of the WoT and DHT overlays and establishing secure communication channels.

Throughout the description of these protocols, $\{M\}_{S_{\mathcal{K}_\mathcal{X}}}$ denotes a message M being signed by user $\mathcal{X}$'s private key $\mathcal{K}_\mathcal{X}^-$, and $E_{\mathcal{K}_\mathcal{Y}}\{M\}$ denotes the message M being encrypted with the user $\mathcal{Y}$'s public key $\mathcal{K}_\mathcal{Y}^+$[2]. The distinct identifiers of users are associated with keypairs: while $\mathcal{N}_\mathcal{X} = \{\mathcal{N}_\mathcal{X}^-, \mathcal{N}_\mathcal{X}^+\}$ denotes the keypair for the Node Id, $\mathcal{U}_\mathcal{X} = \{\mathcal{U}_\mathcal{X}^-, \mathcal{U}_\mathcal{X}^+\}$ denotes the keypair for the User Id, and $\mathcal{W}_\mathcal{X} = \{\mathcal{W}_\mathcal{X}^-, \mathcal{W}_\mathcal{W}^+\}$ denotes the keypair for the random walk Id of node $\mathcal{X}$.

[2]More precisely, session keys are used to encrypt the payload. Such keys are advertised at the beginning of the message encrypted with the target Node Id public key.

8.4.1 Account creation

In order to create an account, a newcomer $\mathcal{V}$ generates a series of keypairs $\mathcal{U}_\mathcal{V}$, $\{\mathcal{N}_{\mathcal{V},0\leq i\leq n}\}$, $\{\mathcal{W}_{\mathcal{V},0\leq j\leq m}\}$ to be associated, respectively, to his personal identity, his n devices and m random walks. All public keys are sent to the TIS, together with $\mathcal{V}$'s identity record $name_\mathcal{V} =< firstName, \ldots, nationality >$ and a proof of identity ownership[3].

Starting from a series of secrets $MK_\mathcal{U}$, $MK_\mathcal{N}$, $MK_\mathcal{W}$, the TIS computes $\mathcal{V}$'s User Identifier $UId_\mathcal{V}$ as a keyed hash function h_{MK} applied to $name_\mathcal{V}$. Similarly, the TIS also computes a set of n node and m random walk identifiers by using $MK_\mathcal{N}$, $MK_\mathcal{W}$ respectively, and applying the keyed hash function on a concatenation of $name_\mathcal{V}$ and an integer $0 \leq p \leq n$.

Finally, each identifier is sent back to $\mathcal{V}$ together with a certificate associating such identifier to a different user-generated public key. The certificate also contains further information on the TIS, a timestamp and an expiration time. Additional meta identifiers are sent back to $\mathcal{V}$. Each identifier is computed as a well known hash function of a possible combination of the values in $name_\mathcal{V}$ such as $h\left(firstName, nationality\right)$.

Once received the certified identifiers, $\mathcal{V}$ can join the P2P network.

Trusted contact establishment. The newcomer $\mathcal{V}$ needs to build his web of trust to access (and provide) Cloud services from his node pool. To find a trusted contact $\mathcal{U}$, $\mathcal{V}$ hashes a subset of properties of $name_\mathcal{U}$ and looks for this meta identifier on the DHT. As an answer, all User Identifiers UId_i associated to such meta identifier are provided to $\mathcal{V}$, which triggers another request for each of them. A list of random walk identifiers WId_{ij} and corresponding auxiliary access points is retrieved for each UId_i. User $\mathcal{V}$ then triggers a profile request to the AAPs. Requests are routed along the random walks thanks to WId_{ij}; publicly available profile data is served by each UId_i (or one of his trusted contacts) and is forwarded back along the same path. At the reception of profile data, $\mathcal{V}$ selects the correct target $UId_\mathcal{U}$ and sends him a contact request containing $\mathcal{V}$'s User- and Nodes- Identifiers together with the TIS certificates and a list of available Cloud services running at $\mathcal{V}$'s node pool. Again, the contact request is sent to $\mathcal{U}$'s AAPs and is routed along the random walks. User $\mathcal{U}$ can accept the request and reply directly to $\mathcal{V}$.

Once a bidirectional trust link has been built, $\mathcal{V}$ can access Cloud services offered by $\mathcal{U}$, and vice-versa.

Cloud service access. The first Cloud service $\mathcal{V}$ accesses is the ***Communication Obfuscation as a Service*** (COaaS), where $\mathcal{V}$ creates a random walk of q hops starting from $\mathcal{U}$. A random walk request RWR message is sent to $\mathcal{U}$ and forwarded along the WoT graph. Such RWR contains a walk token $WTok$, a recursively signed Time To Live message and a signed random number $rnd_{S_{\mathcal{W}^-}}$. The $WTok$ contains the jth random walk Id certificate $Cert\left(WId_{\mathcal{V},j}\right)$ and an expiration time signed with $\mathcal{W}^-$. At each hop, a user in the random walk decreases the TTL and selects a random contact to forward the request. When $TTL = 0$, the current node $\mathcal{G}$ verifies the signature on the random number is associated to $Cert\left(WId_{\mathcal{V},j}\right)$, and registers the pair $\langle DHTkey, DHTvalue\rangle$ on the DHT, where $DHTkey = WId_{\mathcal{V},j}$ and $DHTvalue = [WTok, Cert\left(NId_\mathcal{G}\right)]\, S_{\mathcal{N}_\mathcal{G}^-}$. The presence of $rnd_{S_{\mathcal{W}^-}}$ in the DHT storage request for $\langle DHTkey, DHTvalue\rangle$ triggered by $\mathcal{G}$ poses as an authorization.

Once such association has been registered, a confirmation is routed back according to $WId_{\mathcal{V},j}$ along the random walk. At the same time, $\mathcal{V}$ stores a new association

$$\left\langle UId_\mathcal{V}, [Cert\left(UId_\mathcal{V}\right), Cert\left(WId_{\mathcal{V},j}\right), exptime]\, S_{\mathcal{U}_\mathcal{V}^-}\right\rangle$$

[3] Such proof can consist of a secret shared OOB after face-to-face identity verification.

in the DHT.

Storage of $\left\langle metaId_{\mathcal{V}}, [Cert\left(metaId_{\mathcal{V}}\right), Cert\left(UId_{\mathcal{V}}\right), exptime]\, S_{u_{\mathcal{V}}^-}\right\rangle$ is optional and may happen at any time.

A series of IaaS, SaaS, PaaS services can be provided, with the user consent, to the user's contacts in addition to the user himself. User $\mathcal{U}$ has n real/virtual nodes in the DHT which form $\mathcal{U}$'s node pool and provide basic services like COaaS and storage. Among such nodes, those with higher resources run an hypervisor and instantiate virtual machines, which may be connected to form more complex virtual data centers.

Within the MapReduce framework, trust-based parallel processing is achieved by splitting problems in sub-problems and distributing them to trusted user maintained nodes. Sub-problems may further be divided and dispatched along the WoT.

As shown in Fig. 8.3, among the series of services $\mathcal{U}$ provides to $\mathcal{V}$, part of them may not be run directly on $\mathcal{U}$'s nodes. $\mathcal{U}$ may in fact advertise services provided to him by his trusted contact $\mathcal{Z}$. In this case, $\mathcal{U}$ acts as a proxy for $\mathcal{V}$. When all $\mathcal{U}$'s nodes are disconnected, $\mathcal{V}$ can still access services running at $\mathcal{Z}$ nodes by contacting $\mathcal{U}$'s AAPs.

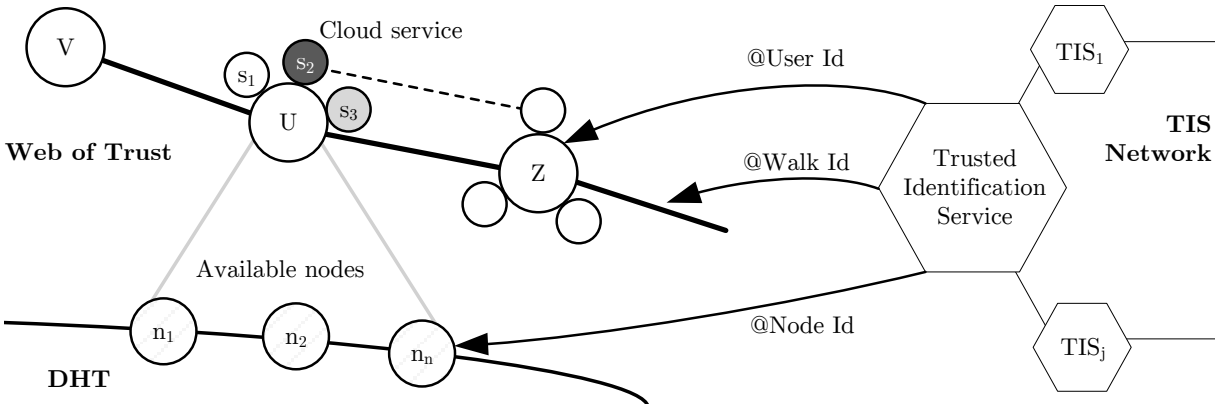


Figure 8.3: Availability of Cloud services provided by user $\mathcal{U}$ to $\mathcal{V}$: in white, available services running at $\mathcal{U}$'s nodes; in black, unavailable ones; in gray, services running at $\mathcal{Z}$ nodes available for $\mathcal{U}$ which $\mathcal{U}$ respectively provides, as a proxy, to $\mathcal{V}$.

8.5 Preliminary Evaluation

A complete performance evaluation of the proposed system is not available at this point, as we are describing work in progress. In the following we however give an overview to evaluate the compliance with respect to the security objectives we required in section 8.2.

Integrity. The one-to-one correspondance between a user's identity and his User Identifier is ensured by the TIS. This prevents malicious users from creating ***fake identities***, or mounting ***impersonation attacks*** and disrupt the WoT. Even when a user starts the account creation process with two different TISs, since the different MK used for identifier computation are shared within the TIS network, such user will receive the same identifiers.

A newcomer $\mathcal{V}$ is prevented from sending a trusted contact request to AAPs which are not those associated to the legitimate target $\mathcal{U}$. The association between a certified meta identifier and a certified UId cannot be stored in the DHT in case of mismatch between the public keys contained in both certificates. The same applies to the association between a UId and a WId.

Finally, the association between a certified WId and a certified AAP Nid cannot be stored without the proof such AAP is at the end of a random walk originated at the legitimate $\mathcal{V}$. Such proof is represented by $rnd_{S_{w-}}$.

Since the association between a walk identifier and the AAP Node Identifier is publicly available, ***collusion*** between DHT nodes and unlegitimate AAPs can be detected and malicious association removed from the DHT.

Privacy. The proposed solution guarantees anonymity, since a user can choose to skip the registration between any of his meta identifiers and his UId. Computation of the victim's UId cannot be performed with ***dictionary attacks*** due to the adoption of a keyed hash function to compute identifiers. Even ***brute force attacks*** to guess a victim's UId fail in case such victim does not provide publicly available information on his identity.

An anonymous user $\mathcal{V}$ may still find and establish a trusted contact with a non anonymous user $\mathcal{U}$. However, when both $\mathcal{V}$ and $\mathcal{U}$ are anonymous users, trusted contact establishment succeeds only after UIds have been exchanged out-of-band. Currently, a TIS can detect if a user is participating in the system or not. We are investigating further distributed authentication mechanisms to overcome this limitation.

Hop-by-hop communication integrity and confidentiality is ensured by signing each message with the sender's node private key and encrypting the concatenation of message and signature with the receiver's node public key. Eavesdroppers are therefore unable to detect any information on parties' identifiers and communication content. Additionaly, in case of trusted contact establishment or asynchronous communication, end-to-end communication integrity and confidentiality is ensured by signature and encryption operations based on keypairs associated to User Identifiers.

In case TCSPs provide storage space, a user $\mathcal{V}$'s data is stored at the TCSP $\mathcal{U}$ in an encrypted and partitioned form. In fact, each user is considered ***honest but curious***.

User identity cannot be linked to IP addresses since the relationship between Node Identifiers and User Identifier is not stored in the DHT and is disclosed to trusted contacts only at the act of contact establishment.

Nodes participating in serving random walks cannot derive any information on the originator of the walk: given a WId, it is impossible to retrieve any information on any other WId related to the same user, consequently, if present, any publicly available information on such user.

Finally, no entity in the system can derive the composition of the WoT, since a user's knowledge of the WoT is limited to his trusted contacts only.

Availability. Adoption of certified identifiers prevents sybil and denial of service attacks. Malicious nodes fail in perpetrating ***DHT poisoning*** due to the presence of signatures in each record they store. Furthermore, due to the parallelism of DHT lookup and storage operations, a malicious peer dropping requests does not impact provisioning of results.

8.6 Related Work

To the best of our knowledge, no work in literature proposes the adoption of decentralized architectures to preserve Cloud users' privacy. On the contrary, a series of work aims at moving towards P2P to increase Cloud service availability.

The work in [XSZS09] proposes a P2P Cloud solution to provide a distributed data storage environment to address the problem of bottlenecks due to central indexes in Cloud storage systems. In Cloud-to-Cloud [GKW11], Cloud service providers participate in a pool of

computer resources. Resource requests which cannot be provisioned by a particular Cloud service provider can be met from such a shared ecosystem of resources in a seamless manner. In [MTT12], authors present the design of P2P MapReduce, an adaptive framework which exploits a P2P model to allow for MapReduce in Cloud-of-clouds environment. Compared to centralized implementations of MapReduce, such solution provides resiliency against node churn and failures.

Researchers also got inspired from P2P systems to propose new way of looking up for resources in Cloud environments. ***Cloud peer*** [RZW+10] creates an overlay network of virtual machines to address service discovery and load-balancing. VM instances update their software and hardware configuration to a DHT supporting indexing and matching of multidimensional range, so that provisioning software can search for and discover them. Authors in [BMT12] propose a distributed IaaS Cloud system using gossip based protocols to manage a pool of peers without coordinators. Users can request a fraction of the available resources matching a given query. Authors in [LY12] propose an hybrid overlay composed by a structured P2P system with an unstructured one to support multi-attribute range query. The overlay is organized in clusters, each cluster being composed by resource groups, in turn composed by peers with the same resources.

The work which is closest to our approach has been presented in Safebook [CMS09], a decentralized architecture for privacy preserving online social networks. As in our solution, Safebook relies on the trust relationships among users to provide P2P services. As opposed to Safebook, our solution does not depend on a single TIS, allows multiple user's nodes to join the network at the same time, and provides higher privacy protection in terms of communication untraceability thanks to the adoption of random walk based routing. Moreover, while Safebook exploits direct contacts resources to provide a limited range of social network services, our solution exploits the entire WoT and is conceived to provide a wide range of Cloud services.

8.7 Conclusion and Future Work

In this chapter, we proposed a decentralized approach to protect critically sensitive user data against the potential control of malicious omniscient Cloud service providers and attackers taking control of them. The approach leverages on existing trust among users to provide Cloud services and anonymize traffic. Our solution consists on a Web of Trust among users, who also play the role of Cloud service providers. User nodes take part in a DHT and run Cloud services, which may be accessed directly or indirectly, as a function of the WoT shape and with the user consent. Resilience against offline Cloud providers is achieved through Auxiliary Access Points for those providers' services. The sensitive information describing WoT users and their relationships is protected through encryption and anonymization techniques similar to onion routing, applied to random walks on the WoT graph.

As future work, we plan to develop a full model of the system in order to study the service availability. We have already started to develop a prototype based on the new WebRTC[4] technology, which allows to build a P2P network from popular web browsers. We plan to complete our prototype and integrate it with popular IaaS solutions such as OpenStack and SaaS such as Hadoop. Since trust between users does not correspond to trust on their hardware, we plan to investigate new mechanisms to guarantee the execution and correctness of a sequence of operations at the Cloud side. Finally, we also plan to evaluate the tradeoff between usage control and anonymity through new distributed collaborative privacy policy enforcement schemes. [CL13]

[4] http://www.webrtc.org

Bibliography

[AA91] D. Agrawal and A. F. Abadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 9(1):1–20, February 1991.

[ABC+02] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, December 2002.

[ABDL07] Nitin Agrawal, Willian J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study o file-system metadata. In *Proc. of the 5th USENIX Conference on File and Storage Technologies – FAST'07*, 2007.

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

[ADD+10] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage. Steward: Scaling byzantine fault-tolerant replication to wide area networks. *IEEE Transactions on Dependable and Secure Computing*, 7(1):80–93, 2010.

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

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

[AI12] Cloud Security Alliance and ISACA. Cloud computing market maturity, 2012. http://www.isaca.org/Knowledge-Center/Research/ResearchDeliverables/Pages/2012-Cloud-Computing-Market-Maturity-Study-Results.aspx.

[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.

[Ama06] Amazon Web Services. Amazon Elastic Compute Cloud (Amazon EC2), 2006. http://aws.amazon.com/ec2/.

[AW96] Y. Amir and A. Wool. Evaluating quorum systems over the internet. In *Proceedings of the 26th Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*, 1996.

[AWS11] AWS Team. Summary of the Amazon EC2 and Amazon RDS service disruption in the US east region. http://aws.amazon.com/message/65648/, 2011.

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

[BAP07] Jeremy Buisson, Francoise Andre, and Jean-Louis Pazat. Supporting adaptable applications in grid resource management systems. In *Proc. of the IEEE/ACM Int. Conf. on Grid Computing*, 2007.

[BBH+11] W. Bolosky, D. Bradshaw, R. Haagens, N. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *In the Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation – NSDI '11*, 2011.

[BCE+12] C. Basescu, C. Cachin, I. Eyal, R. Haas, A. Sorniotti, M. Vukolic, and I. Zachevsky. Robust data sharing with key-value stores. In *Proceedings of the 42nd International Conference on Dependable Systems and Networks - DSN 2012*, June 2012.

[BCQ+11] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. In *Proc. of ACM EuroSys'11*, 2011.

[Bec09] C. Beckmann. Google app engine: Information regarding 2 july 2009 outage. https://groups.google.com/forum/?fromgroups=#!msg/google-appengine/6SN_x7CqffU/ecHIgNnelboJ, 2009.

[Bes11] Alysson Bessani. From Byzantine fault tolerance to intrusion tolerance (a position paper). In *Proc. of the 5th Workshop on Recent Advances in Intrusion-Tolerant Systems – WRAITS'11 (together with IEEE/IFIP DSN'11)*, July 2011.

[BGC11] Rajkumar Buyya, Saurabh Garg, and Rodrigo Calheiros. SLA-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions. In *Proc. of Cloud and Service Computing*, 2011.

[BK02] Omar Bakr and Idit Keidar. Evaluating the running time of a communication round over the internet. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 243–252, 2002.

[BK08] Omar Bakr and Idit Keidar. On the performance of quorum replication on the internet. Technical report, EECS Department, University of California, Berkeley, Oct 2008.

[BKSS11] Peter Bokor, Johannes Kinder, Marco Serafini, and Neeraj Suri. Efficient model checking of fault-tolerant distributed protocols. In *Proc. of the 41st IEEE Int'l Conf. on Dependable Systems and Networks – DSN'11*, 2011.

[BMT12] Ozalp Babaoglu, Moreno Marzolla, and Michele Tamburini. Design and implementation of a P2P cloud system. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 412–417, Trento, Italy, March 2012. ACM.

[BSA] Alysson Bessani, Joao Sousa, and Eduardo Alchieri. BFT-SMaRt: High-performance Byzantine-Fault-Tolerant State Machine Replication. *http://code.google.com/p/bft-smart*.

[BSF+13] Alysson Bessani, Marcel Santos, Joao Felix, Nuno Neves, and Miguel Correia. On the efficiency of durable state machine replication. In *Proc. of the USENIX Annual Technical Conference – USENIX ATC 2013*, June 2013.

[BT12] Cory Bennett and Ariel Tseitlin. Chaos monkey released in the wild. http://techblog.netflix.com/2012/07/chaos-monkey-\released-into-wild.html, 2012.

[Bur06] Mike Burrows. The Chubby lock service. In *Proceedings of 7th Symposium on Operating Systems Design and Implementation – OSDI 2006*, November 2006.

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

[Cal11] B. Calder et. al. Windows Azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles – SOSP'11*, 2011.

[CGR07] Tushar Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live - an engineering perspective (2006 invited talk). In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing - PODC'07*, August 2007.

[Cho12] S. Choney. Amazon Web Services outage takes down Netflix, other sites. http://www.nbcnews.com/technology/technolog/amazon-web-services-outage-takes-down-netflix-\other-sites-1C6611522, 2012.

[CHS01] W. K. Chen, M. Hiltunen, and R. Schlichting. Constructing adaptive software in distributed systems. In *Proceedings of the 21th IEEE International Conference on Distributed Computing Systems*, April 2001.

[CKL+09] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 277–290, 2009.

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

[CL13] Leucio Antonio Cutillo and Antonio Lioy. Towards privacy-by-design peer-to-peer cloud computing. In *10th International Conference on Trust, Privacy & Security in Digital Business*, TrustBus '13, Prague, Czech Republic, 2013.

[CLM+08] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, 2008.

[Clo10] Cloud Security Alliance. Top threats to cloud computing v1.0, March 2010.

[CMS09] L.A. Cutillo, R. Molva, and T. Strufe. Safebook: A privacy-preserving online social network leveraging on real-life trust. *IEEE Communications Magazine*, 47(12):94–101, December 2009.

[CNS+13] Vinicius Vielmo Cogo, Andre Nogueira, Joao Sousa, Marcelo Pasin, Hans P. Reiser, and Alysson Bessani. FITCH: Supporting adaptive replicated services in the cloud. In *Proc. of the 13th IFIP International Conference on Distributed Applications and Interoperable Systems – DAIS'13*, June 2013.

[Com13a] Comscore. How technology and analytics drive the mobile market, February 2013. http://www.comscore.com/Insights/Presentations_and_Whitepapers/2013/How_Technology_and_Analytics_Drive_the_Mobile_Market.

[Com13b] Comscore. The rise of big data and the internet, January 2013. http://www.comscore.com/Insights/Presentations_and_Whitepapers/2013/The_Rise_of_Big_Data_on_the_Internet.

[Cor12] J. Corbett et al. Spanner: Google's globally-distributed database. In *Proceedings of 10th Symposium on Operating Systems Design and Implementation – OSDI 2012*, 2012.

[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.

[CRM91] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, pages 181–186, New York, NY, USA, 1991.

[CS10] Yao Chen and Radu Sion. On securing untrusted clouds with cryptography. In *Proceedings of the 9th annual ACM workshop on Privacy in the electronic society*, WPES '10, pages 109–114, Chicago, Illinois, USA, October 2010. ACM.

[CST+10] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, 2010.

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

[Dea09] J. Dean. Google: Designs, lessons and advice from building large distributed systems. In *Keynote at the ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, October 2009.

[Dek12] M. A. C. Dekker. Critical Cloud Computing: A CIIP perspective on cloud computing services (v1.0). Technical report, European Network and Information Security Agency (ENISA), December 2012.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 205–220, 2007.

[DKO+84] David DeWitt, Randy Katz, Frank Olken, Leonard Shapiro, Michael Stonebraker, and David Wood. Implementation techniques for main memory database systems. In *Proc. of the ACM Int. Conf. on Management of Data*, 1984.

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

[DMM+12] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A Pras. Inside Dropbox: Understanding personal cloud storage services. In *Proc. of the 12th ACM Internet Measurement Conference - IMC'12*, Nov 2012.

[DPC10] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. Autonomous resource provisioning for multi-service web applications. In *Proc. of the WWW*, 2010.

[DPSP+11] Tobias Distler, Ivan Popov, Wolfgang Schröder-Preikschat, Hans P. Reiser, and Rüdiger Kapitza. SPARE: Replicas on hold. In *Proceedings of the 18th Network and Distributed System Security Symposium*, pages 407–420, February 2011.

[EC2a] Amazon EC2 instance types. http://aws.amazon.com/ec2/instance-types/.

[EC2b] Amazon EC2 pricing. http://aws.amazon.com/ec2/pricing/.

[EH-] Elastichosts cloud servers pricing. http://www.elastichosts.com/cloud-servers-quote/.

[Fil] Filebench webpage. http://sourceforge.net/apps/mediawiki/filebench/.

[fir12] Firebird project. firebirdsql, April 2012.

[FLP+10] D. Ford, F. Labelle, F. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[FUS] FUSE: File system in user space. http://fuse.sourceforge.net.

[Gar04] Garlan, D. *et al.* Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10), 2004.

[Gar12] Gartner. Forecast overview: Public cloud services, worldwide, 2011-2016, 2q12 update, August 2012. http://www.gartner.com/id=2126916.

[GBG+13] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. Analysis of OS diversity for intrusion tolerance. *Software - Practice and Experience*, 2013. to appear.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, January 1985.

[Gif79] David Gifford. Weighted voting for replicated data. In *Proc. of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, December 1979.

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

[GKW11] Ankur Gupta, Lohit Kapoor, and Manisha Wattal. C2c (cloud-to-cloud): An ecosystem of cloud service providers for dynamic resource provisioning. In Ajith Abraham, Jaime Lloret Mauri, JohnF. Buford, Junichi Suzuki, and SabuM. Thampi, editors, *Advances in Computing and Communications*, volume 190 of *Communications in Computer and Information Science*, pages 501–510. Springer Berlin Heidelberg, 2011.

[GMB85] Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, October 1985.

[GMS92] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, December 1992.

[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.

[GOO] Google cloud storage access control. http://developers.google.com/storage/docs/accesscontrol.

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

[GRP11] Rui Garcia, Rodrigo Rodrigues, and Nuno Preguica. Efficient middleware for Byzantine fault-tolerant database replication. In *Proc. of ACM EuroSys'11*, 2011.

[Gru03] Andreas Grunbacher. POSIX access control lists on linux. In *Proc. of the USENIX Annual Technical Conference – ATC 2003*, June 2003.

[GS-] Google storage pricing. https://developers.google.com/storage/docs/pricingandterms.

[Ham12] J. Hamilton. Observations on errors, corrections, and trust of dependent systems. http://perspectives.mvdirona.com/2012/02/26/ObservationsOnErrorsCorrectionsTrustOfDependentSystems.aspx, 2012.

[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.

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

[HDV+11] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles – SOSP'11*, October 2011.

[Her87] Maurice Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Transactions on Database Systems*, 12(2):170–194, June 1987.

[HKJR10] 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.

[HKKF95] Y. Huang, C. Kintala, N. Kolettis, and N.D. Fulton. Software rejuvenation: analysis, module and applications. In *Proc. of FTCS*, 1995.

[HKM+88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

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

[HW90] 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.

[hyp12] Hypersql, hsqldb, April 2012.

[JBo10] JBoss Community. Netty - the java nio client-server socket framework. *http://www.jboss.org/netty*, 2010.

[JRS11] F. Junqueira, B. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proc. of the Int. Conf. on Dependable Systems and Networks*, 2011.

[KA08] Jonathan Kirsh and Yair Amir. Paxos for system builders: An overview. In *LADIS*, 2008.

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

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

[KBB01] B. Kemme, A. Bartoli, and O. Babaoglu. Online reconfiguration in replicated databases based on group communication. In *Proc. of the Int. Conf. on Dependable Systems and Networks*, 2001.

[KBC+00] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, R. Gummadi D. Geels, S. Rhea, H. Weatherspoon, C. Wells W. Weimer, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[KBC+12] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: resource-efficient Byzantine fault tolerance. In *Proc. of ACM EuroSys'12*, 2012.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1), 2003.

[KZHR07] Ruediger Kapitza, Thomas Zeman, Franz Hauck, and Hans P. Reiser. Parallel state transfer in object replication systems. In *DAIS*, 2007.

[LAB+06] Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The SMART way to migrate replicated stateful services. In *Proc. of ACM EuroSys'06*, pages 103–115, April 2006.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.

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

[Lam91] Kwok-Yan Lam. An implementation for small databases with high availability. *SIGOPS Operating Systems Rev.*, 25(4), October 1991.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions Computer Systems*, 16(2):133–169, May 1998.

[Les01] Leslie. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[LFKA11] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proc. 23rd ACM Symposium on Operating Systems Principles – SOSP'11*, Cascais, Portugal, October 2011.

[LGG+91] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Liuba Shrira. Replication in the Harp file system. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 1991.

[LKMS04] Jinyuan Li, Maxwell N. Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proc. of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, December 2004.

[LMZ10a] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, March 2010.

[LMZ10b] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1), 2010.

[LPGM08] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proc. of the USENIX Annual Technical Conference – ATC 2008*, June 2008.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, July 1982.

[LY12] Kuan-Chou Lai and You-Fu Yu. A scalable multi-attribute hybrid overlay for range queries on the cloud. *Information Systems Frontiers*, 14(4):895–908, 2012.

[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/.

[Mil68] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference*, AFIPS '68 (Fall, part I), pages 267–277, 1968.

[Mil08] R. Miller. Explosion at The Planet causes major outage. *Data Center Knowledge*, June 2008.

[MJM08] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 369–384, 2008.

[Mon] Mongodb. http://www.mongodb.org.

[MSL+10] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. In *Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation – OSDI 2010*, pages 307–322, October 2010.

[MTT12] Fabrizio Marozzo, Domenico Talia, and Paolo Trunfio. P2P-MapReduce: Parallel data processing in dynamic cloud environments. *Journal of Computer and System Sciences*, 78(5):1382–1402, September 2012. JCSS Special Issue: Cloud Computing 2011.

[mys12] Mysql, April 2012.

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

[Nie93] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers, 1993.

[NW98] M. Naor and A. Wool. Access control and signatures via quorum secret sharing. *IEEE Transactions on Parallel and Distributed Systems*, 9(9):909–922, 1998.

[ORS+11] Diego Ongaro, Stephen M. Ruble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2011.

[pos12] Postgresql, April 2012.

[Pri] Marco Primi. libpaxos[2]. http://libpaxos.sourceforge.net/.

[PS02] Jaehong Park and Ravi Sandhu. Towards usage control models: beyond traditional access control. In *Proceedings of the seventh ACM Symposium on Access Control Models and Technologies*, SACMAT '02, pages 57–64, Monterey, California, USA, June 2002. ACM.

[PW97] David Peleg and Avishai Wool. Crumbling walls: a class of practical and efficient quorum systems. *Distributed Computing*, 10:87–97, 1997.

[Pâr86] Jehan Pâris. Voting with witnesses: A consistency scheme for replicated files. In *In Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 3–6, 1986.

[Rap11] J.R. Raphael. The 10 worst cloud outages (and what we can learn from them). Infoworld. Available at http://www.infoworld.com/d/cloud-computing/the-10-worst-cloud-outages-and-what-we-can-learn-them-902, 2011.

[Ric11] M. Ricknas. Lightning strike in Dublin downs Amazon, Microsoft clouds. *PC World*, August 2011.

[RK07] Hans Reiser and Rudiger Kapitza. Hypervisor-based efficient proactive recovery. In *Proc. of SRDS*, 2007.

[RS-a] Rackspace cloud files pricing. http://www.rackspace.com/cloud/files/pricing/.

[RS-b] Rackspace cloud servers pricing. http://www.rackspace.com/cloud/servers/pricing/.

[RST11] Jun Rao, Eugene J. Shenkita, and Sandeep Tata. Using Paxos to build a scalable, consistent, and highly available datastore. *Proceedings of the VLDB Endowment*, 2011.

[RZW+10] Rajiv Ranjan, Liang Zhao, Xiaomin Wu, Anna Liu, Andres Quiroz, and Manish Parashar. Peer-to-peer cloud provisioning: Service discovery and load-balancing. In Nick Antonopoulos and Lee Gillam, editors, *Cloud Computing*, Computer Communications and Networks, pages 195–217. Springer London, 2010.

[S3-] Amazon S3 pricing. http://aws.amazon.com/s3/pricing/.

[S3A] Amazon S3 access control list (ACL) overview. http://docs.aws.amazon.com/AmazonS3/latest/dev/ACLOverview.html.

[S3F] S3FS - FUSE-based file system backed by Amazon S3. http://code.google.com/p/s3fs/.

[S3Qa] S3QL - a full-featured file system for online data storage. http://code.google.com/p/s3ql/.

[S3Qb] S3QL 1.13.2 documentation: Known issues. http://www.rath.org/s3ql-docs/issues.html.

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

[SB12] Joao Sousa and Alysson Bessani. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *Proceedings of the 9th European Dependable Computing Conference – EDCC'12*, 2012.

[SBC+10] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, 2010.

[SBG10] Jiri Simsa, Randy Bryant, and Garth Gibson. dbug: Systematic evaluation of distributed systems. In *Proc. of the 5th Int. Workshop on Systems Software Verification – SSV'10*, 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.

[SK11] S. Subashini and V. Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications*, 34(1):1–11, 2011.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proc of the 26th IEEE Symposium on Massive Storage Systems and Technologies – MSST'10*, 2010.

[SNI12] SNIA. Cloud data management interface (CDMI), version 1.0.2. SNIA Technical Position, June 2012.

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

[SSZ+09] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, M.Frans Kaashoek, and Robert Morris. Flexible, Wide-Area Storage for Distributed System with WheelFS. In *In the Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation – NSDI '09*, April 2009.

[Sun04] Sun Microsystems. Web services performance: Comparing Java™ 2 enterprise edition (J2EE™ platform) and .NET framework. Technical report, Sun Microsystems, Inc., 2004.

[SUR] 2012 future of cloud computing - 2nd annual survey results. http://mjskok.com/resource/2012-future-cloud-computing-2nd-annual-survey-results.

[SvDJO12] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. Iris: a scalable cloud file system with efficient integrity checks. In *Proc. of the 28th Annual Computer Security Applications Conference - ACSAC '12*, 2012.

[SZ05] Fred B. Schneider and Lidong Zhou. Implementing trustworthy services using replicate state machines. *IEEE Security & Privacy*, 3(5):34–43, September 2005.

[TBZS11] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer. Benchmarking file system benchmarking: It *IS* rocket science. In *Proc. of the 13th USENIX Workshop on Hot Topics in Operating Systems – HotOS XIII*, Napa, CA, May 2011.

[TJWZ08] A. Traeger, N. Joukov, C. P. Wright, and E. Zadok. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage*, 4(2):25–80, May 2008.

[tpc12] Transaction processing performance council, April 2012.

[VBLM07] Benjamin Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating Byzantine faults in database systems using commit barrier scheduling. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, October 2007.

[VCB+13] Giuliana Veronese, Miguel Correia, Alysson Bessani, Lau Lung, and Paulo Verissimo. Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, 62(1), 2013.

[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 International Symposium on Reliable Distributed Systems*, pages 135–144, 2009.

[VCBL10] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. EBAWA: Efficient Byzantine agreement for wide-area networks. In *Proceedings of the IEEE 12th International Symposium on High-Assurance Systems Engineering*, pages 10–19, November 2010.

[Ver02] Paulo Verissimo. Travelling through wormholes: Meeting the grand challenge of distributed systems. In *Proceedings of the International Workshop on Future Directions in Distributed Computing - FuDiCo 2002*, pages 144–151. Springer-Verlag, June 2002.

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

[VSV12] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. BlueSky: A cloud-backed file system for the enterprise. In *Proc. of the 10th USENIX Conference on File and Storage Technologies – FAST'12*, 2012.

[Vuk10] Marko Vukolić. The Byzantine empire in the intercloud. *ACM SIGACT News*, 41:105–111, September 2010.

[WA-] Windows Azure pricing. http://www.windowsazure.com/en-us/pricing/details/.

[WAD12] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating data and metadata for efficient and available storage replication. In *Proc. of the USENIX Annual Technical Conference*, 2012.

[WB90] Samuel D. Warren and Louis D. Brandeis. The right to privacy. *Harward Law Review*, 4(5):193–220, December 1890.

[WBM+06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 307–320, 2006.

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

[WUA+08] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proc. of the 6th USENIX Conference on File and Storage Technologies – FAST'08*, 2008.

[XSZS09] Ke Xu, Meina Song, Xiaoqi Zhang, and Junde Song. A cloud computing platform based on P2P. In *IEEE International Symposium on IT in Medicine & Education*, volume 1 of *ITIME '09*, pages 427–432, Jinan, Shandong, China, August 2009.

[YAK11] Sangho Yi, Artur Andrzejak, and Derrick Kondo. Monetary cost-aware checkpointing and migration on amazon cloud spot instances. *IEEE Trans. on Services Computing*, PP(99), 2011.

[Zha00] Wensong Zhang. Linux virtual server for scalable network services. In *Proc. of Linux*, 2000.