D3.2
Specification of security enablers for data management

Project number:	643964
Project acronym:	**SUPERCLOUD**
Project title:	User-centric management of security and dependability in clouds of clouds
Project Start Date:	1^{st} February, 2015
Duration:	36 months
Programme:	H2020-ICT-2014-1
Deliverable Type:	Report
Reference Number:	ICT-643964-D3.2/ 1.0
Work Package:	WP 3
Due Date:	Nov 2016 - M22
Actual Submission Date:	29^{th} November, 2016
Responsible Organisation:	IBM
Editor:	Marko Vukolić
Dissemination Level:	PU
Revision:	1.0
Abstract:	This deliverable introduces the processing functions for data management in the SUPERCLOUD. In particular, it contains security and dependability component specifications, descriptions of distributed protocols, specifications of cryptographic mechanisms, and descriptions of the data-resilience tools.
Keywords:	data management, dependability, multi-cloud, security.

This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 643964.

This work was supported (in part) by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contract number 15.0091.

Editor

Marko Vukolić (IBM)

Contributors (ordered according to beneficiary numbers)

Mario Münzer (TEC)
Sébastien Canard, Marie Paindavoine (ORANGE)
Andre Nogueira, Antonio Casimiro, João Sousa, Joel Alcântara, Tiago Oliveira, Ricardo Mendes, Alysson Bessani (FFCUL)
Christian Cachin, Simon Schubert (IBM)
Caroline Fontaine (IMT)
Daniel Pletea, Meilof Veeningen (PEN)
Jialin Huang (TUDA)

Disclaimer

Executive Summary

This deliverable introduces the processing functions for data management in the SUPERCLOUD. In particular, it contains security and dependability component specifications, descriptions of distributed protocols, specifications of cryptographic mechanisms, and descriptions of the data-resilience tools.

SUPERCLOUD data management components described in this deliverable are organized in three parts. The first part describes novel components pertaining to state-machine replication, which will be used to replicate critical pieces of SUPERCLOUD metadata across multiple clouds. The second part covers SUPERCLOUD distributed storage solutions which we use to manage bulk data. Finally, the third part describes advanced data security components, focusing, in particular on data privacy techniques. These techniques can be optionally combined with dependability and security components pertaining to fault-tolerant state-machine replication and distributed storage described in the first two parts. Several components described in this deliverable have already been published in top research conferences.

More specifically, the first part of this deliverable focuses on novel solutions for state-machine replication. SUPERCLOUD state-machine replication will use Hyperledger fabric open-source blockchain as its envelope, which is first discussed in Chapter 2. Then, we introduce novel distributed protocols for state-machine replication developed in the context of SUPERCLOUD. Namely, Chapter 3 discusses how to treat non-determinism when replicating arbitrary applications when replicas can fail in an arbitrary (i.e., Byzantine) way. Chapter 4 introduces a novel model for developing reliable distributed protocols called XFT, as well as the first state-machine replication protocol in this model - XPaxos. Chapter 5 empirically evaluates latency-optimization for state-machine replication in WANs and informs the design of novel state-machine replication protocols. Chapter 6 introduces a generic state-transfer tool for partitioned state-machine replication that enables elasticity.

In the second part of the deliverable, we turn to resilient distributed storage. Chapter 7 describes Janus, a multi-cloud storage platform that finds the best way to store data in the clouds according to given user-defined requirements. In Chapter 8 we present new erasure-coded storage emulations on top of untrusted cloud storage services that support multiple concurrent writers. Chapter 9 concludes the set of novel dependability components by presenting a solution for cloud-based database disaster-recovery.

Finally, the third part deals with advanced techniques for enduring data privacy and confidentiality, as well as security of data sharing and anonymization. In particular, Chapter 10 proposes a privacy-preserving distributed solution for verifiable computation. Chapter 11 proposes a solution for privacy-preserving image processing relevant for SUPERCLOUD healthcare use cases. Chapter 12 introduces further privacy preserving techniques based on key encapsulation, proxy re-encryption and attribute-based encryption, and includes details about secure deduplication. Finally, Chapter 13 presents our anonymization techniques.

The deliverable further specifies the integration vectors of the described components and explains how these components come together within the overall WP3 architecture as defined in D3.1. This informs the integration of the (subset of) components that will be tackled on in the following months, in the context of SUPERCLOUD deliverable D3.3.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

SUPERCLOUD WP3 deals with protection of user data and assets in the distributed cloud. The focus of our work is on autonomic and end-to-end security as well as dependability, but also performance and cost. This WP provides a common user experience of data protection across multiple underlying clouds through modular and on-demand data security services such as blind computation, integrity and verifiability, and data availability.

This deliverable details SUPERCLOUD data management security and dependability components. It gives specifications of these components and the description of distributed protocols and cryptographic mechanisms these components are built upon. In a nutshell, this deliverable presents the approach to implementing the solutions for challenges described in deliverable D3.1, in the context of the SUPERCLOUD data management architecture which is itself described in D3.1.

1.1 Deliverable Organization

This deliverable is orchestrated in three parts. The first part of the deliverable deals with state-machine replication in SUPERCLOUD. As described in more details in Section 1.3, state-machine replication will serve in SUPERCLOUD as a technique for replicating critical services across multiple clouds and geographical locations (e.g., metadata management services). This part of the deliverable is organized as follows:

- SUPERCLOUD state-machine replication will leverage Hyperledger fabric open-source blockchain as its envelope, which is discussed in Chapter 2. In the subsequent chapters, we introduce novel distributed protocols for state-machine replication developed in the context of SUPERCLOUD.
- Chapter 3 discusses how to treat non-determinism when replicating arbitrary applications when replicas can fail in an arbitrary (i.e., Byzantine) way.
- Chapter 4 introduces a novel model for developing reliable distributed protocols called XFT, as well as the first state-machine replication protocol in this model - XPaxos.
- Chapter 5 empirically evaluates latency-optimization for state-machine replication in WANs and informing the design of novel state-machine replication protocols.
- Chapter 6 introduces a generic state-transfer tool for partitioned state-machine replication that enables elasticity.

In the second part of the deliverable, we turn to resilient distributed storage components. These components provide secure and dependable way to managing bulk data of SUPERCLOUD users. This part of the deliverable is organized as follows:

- Chapter 7 describes Janus, a multi-cloud storage platform that finds the best way to store data in the clouds according to given user-defined requirements.

- In Chapter 8 we present new erasure-coded storage emulations on top of untrusted cloud storage services that support multiple concurrent writers.
- Finally, Chapter 9 concludes the set of novel dependability components by presenting a solution for cloud-based database disaster-recovery solution.

The third part of this deliverable details advanced privacy and confidentiality components for SUPERCLOUD data management. The components in this part also focus on security of data sharing and data anonimization. In particular, in this part we detail the following components

- Chapter 10 proposes a privacy-preserving distributed solution for verifiable computation.
- Chapter 11 proposes a solution for privacy-preserving image processing relevant for SUPERCLOUD healthcare use cases.
- Chapter 12 introduces further privacy preserving techniques based on key encapsulation, proxy re-encryption and attribute-based encryption, and includes details about secure deduplication.
- Finally, Chapter 13 presents our anonymization techniques.

1.2 Publications and impact

It is worth noting that many of the proposed components have been already presented and published in prestigious peer-reviewed conferences and are already gaining significant traction. Notably:

- Chapter 3 is based on

  Christian Cachin, Simon Schubert, Marko Vukolić: Non-determinism in Byzantine fault-tolerant replication. OPODIS 2016. To appear.

- Chapter 4 is based on

  Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, Marko Vukolić: XFT: Practical Fault-Tolerance Beyond Crashes. OSDI 2016: 485-500.

- Chapter 5 is based on

  João Sousa, Alysson Bessani: Separating the WHEAT from the Chaff: An Empirical Design for Geo-Replicated State Machines. SRDS 2015: 146-155

- Chapter 8 is based on

  Tiago Oliveira, Ricardo Mendes and Alysson Bessani: Exploring Key-Value Stores in Multi-Writer Byzantine-Resilient Register Emulations. OPODIS 2016. To appear.

- Chapter 10 is based on

  Berry Schoenmakers, Meilof Veeningen, Niels de Vreede: Trinocchio: Privacy-Preserving Outsourcing by Distributed Verifiable Computation. ACNS 2016: 346-366

1.3 Component integration

Deliverable D3.1 defined the high-level architecture of WP3. In this deliverable we describe some of the individual security and dependability components that are developed in WP3. In the rest of this introductory chapter, we first recall the high-level WP3 architecture from D3.1 (Chapter 1.3.1), and then explain integration vectors available to components described in this deliverable.

Notice that not all WP3 components described in this deliverable will be integrated — the choice of components that will actually be integrated will be specified in deliverables D3.3 (M28) and D3.4 (M36).

1.3.1 Review: high-level WP3 architecture

A view of the logical architecture of the SUPERCLOUD data management layer, as defined in Deliverable D3.1, is shown in Figure 1.1.

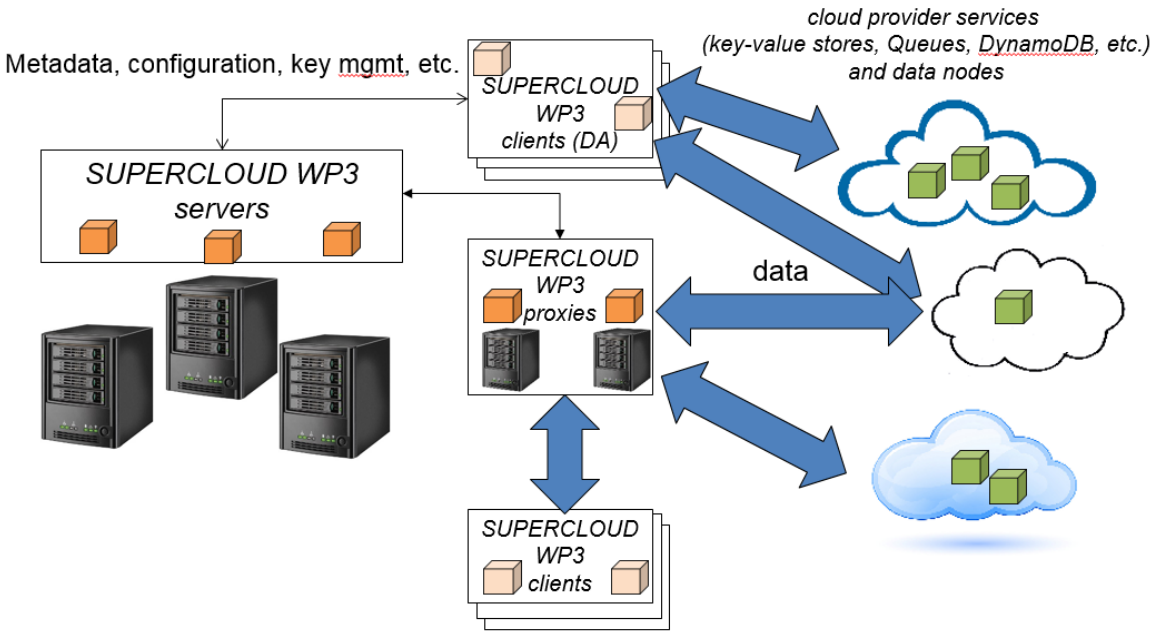


Figure 1.1: High level logical architecture of the SUPERCLOUD data management layer.

For completeness, we recall the main five classes of entities described in D3.1:

- Storage clients (or simply clients), users of SUPERCLOUD storage infrastructure. These are all L2 virtual machines (or containers) deployed in the User clouds. Other storage entities may or may not act as storage clients depending on whether they are deployed as L1 virtual machines, in which case they use cloud provider storage or L2 virtual machines in which case they act as storage clients.

  Clients are divided into two subcategories: direct accessors (DA) and ordinary clients. Ordinary clients interact with the SUPERCLOUD data management layer via storage proxy and are entirely oblivious the SUPERCLOUD data management layer. This requires minimum changes to clients without installing additional libraries. In contrast, DA clients run SUPERCLOUD specific logic as a client library and can interact and access directly storage servers and L1 cloud provider services. DA clients can also have certain functionality of storage servers built-in, making them also possibly independent of storage servers.

- Storage proxies (or simply proxies), are L2 virtual machines (or containers).Proxies are in principle stateless, and can be easily added dynamically to the system. Their primary goal is facilitating clients' access to SUPERCLOUD storage and data management offerings. As we will detail later, examples of proxies include encryption proxies, secure deduplication proxies, etc.

- Storage servers (or simply servers) are stateful L1 or L2 virtual machines (or containers). The main role of servers is housekeeping of critical portions of metadata vital to operation of the

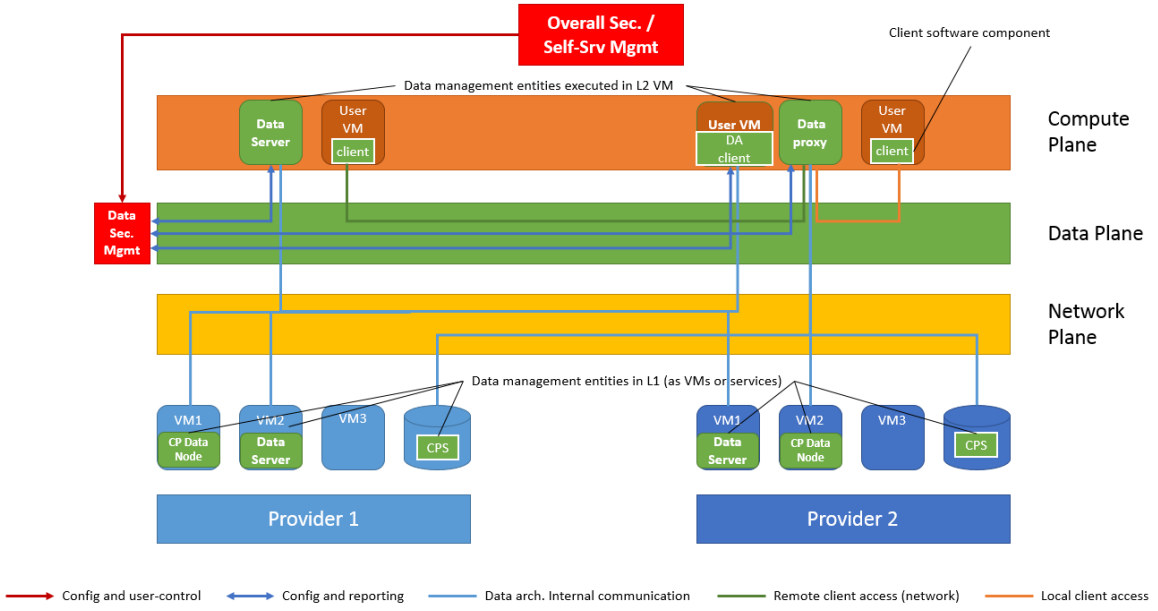


Figure 1.2: Mapping of SUPERCLOUD data management entities to L1 and L2 virtualization layers.

SUPERCLOUD data management layer. Examples of SUPERCLOUD storage servers includes storage metadata servers, data integrity servers, configuration management servers, etc.

- Cloud provider services (CPS) are L1 cloud storage services that DA clients or proxies can directly access. They expose different APIs notably object storage and block storage. Examples include Openstack Swift, Amazon S3 and EBS, etc.
- Cloud provider data nodes (or simply data nodes) are L1 virtual machines or containers that reside on L1 public or private clouds comprising SUPERCLOUD infrastructure. Complementing CPS, data nodes can perform computation and have locally mounted L1 block storage that ends up storing SUPERCLOUD user data.

Mapping of SUPERCLOUD storage and data management entities to L1 and L2 SUPERCLOUD virtualization layers is shown in Figure 1.2.
In the following section, we explain how components described in this deliverable fit into the WP3 architecture.

1.3.2 Prospective Integration Vectors

We identify the following prospective vectors for integration of SUPERCLOUD data management components identified in this deliverable:

- **State machine replication**. This vector orchestrates state machine replication components around open source projects Hyperledger fabric [263] and BFT-SMaRt [5]. It is deployed across SUPERCLOUD storage servers and cloud provider services.
- **Resilient storage**. This vector orchestrates resilient multi-cloud and disaster recovery solutions around the Janus User-Defined Storage. It is deployed across SUPERCLOUD clients (DAs) and cloud provider data nodes and cloud provider services.
- **Privacy-preserving components**. This vector orchestrates security and, in particular, privacy-preserving data management components. It is deployed across SUPERCLOUD clients and storage proxies.

In the following, we overview each of the proposed prospective integration vectors and discuss how they fit together in the overall WP3 architecture.

1.3.2.1 State-machine replication

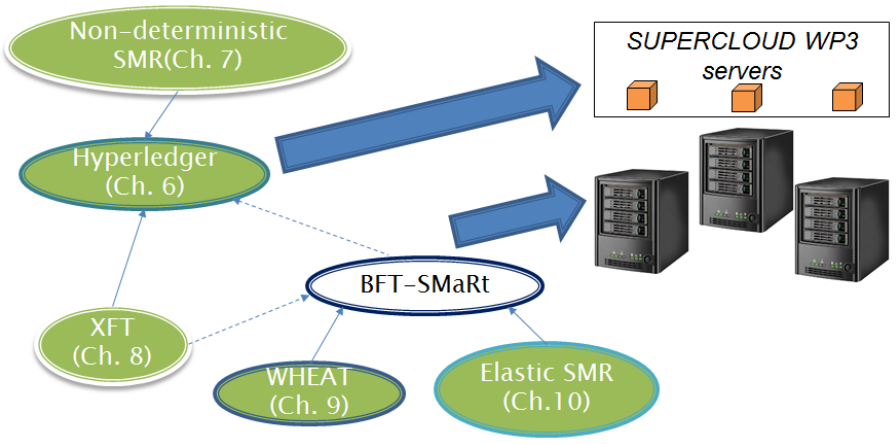


Figure 1.3: State-machine replication integration vector and its fit within the SUPERCLOUD data management architecture. Green components are described in this deliverable in respective chapters. Full lines represent single-partner integration vectors, dashed lines represent integration vectors across two or more partners.

This integration vector is foreseen for the following state-machine replication (SMR) components of SUPERCLOUD:

- Protocols that deal with SMR non-determinism, such as Sieve protocol, described in Chapter 3,
- XFT protocols, described in Chapter 4,
- WHEAT geo-replication, described in Chapter 5, and
- Elastic SMR, described in Chapter 6.

The integration of a subset or all of these components will be made possible through open source projects Hyperledger fabric described in Chapter 2 and the BFT-SMaRt open-source project maintained by FCUL [5]. This integration vector primarily orchestrates SUPERCLOUD storage servers. Possible integration is depicted in Figure 1.3.

1.3.2.2 Resilient distributed storage

This integration vector is foreseen for the following multi-cloud resilience components of SUPERCLOUD:

- Janus, user-centric multi-cloud storage, described in Chapter 7,
- Resilient multi-writer storage component, described in Chapter 8,
- Ginja disaster recovery component, described in Chapter 9.
- Secure deduplication component, described in Chapter 12, Section 12.2.

We plan to integrate a subset of these components as SUPERCLOUD data management clients (DAs), SUPERCLOUD data nodes and cloud provider services. Some components will be integrated in the Janus User-Defined Cloud Storage, a component under development by FCUL. Possible integration is depicted in Figure 1.4.

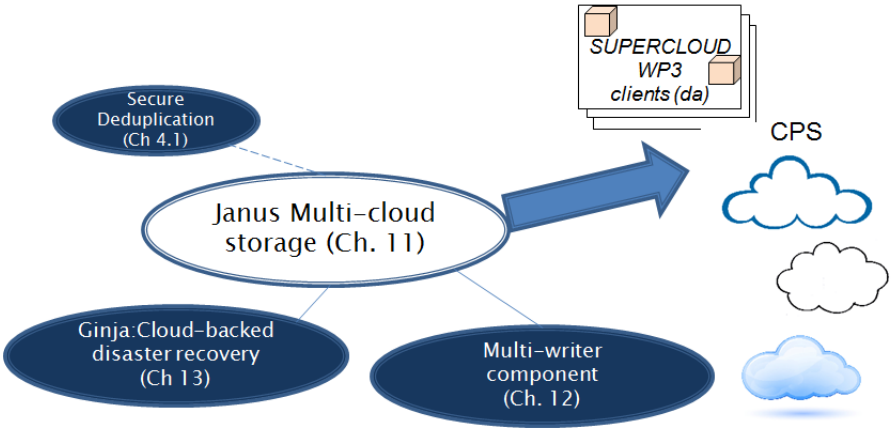


Figure 1.4: Resilient distributed storage integration vector and its fit within the SUPERCLOUD data management architecture. Green components are described in this deliverable in respective chapters. Full lines represent single-partner integration vectors, dashed lines represent integration vectors across two or more partners.

1.3.2.3 Advanced privacy-preserving components

Whereas some components developed in the context of state-machine replication and resilient distributed storage cover some aspects of confidentiality, we specifically focus on advanced privacy-preserving components which represents also our third integration vector. This integration vector is foreseen for the following privacy-preserving components of SUPERCLOUD:

- Trinocchio, distributed verifiable computation component, described in Chapter 10,
- Privacy of image processing, described in Chapter 11,
- Proxy re-encryption and attribute based encryption components, described in Chapter 12, and
- K-anonymity component, described in Chapter 13.

We plan to integrate a subset of these components as SUPERCLOUD data management clients and SUPERCLOUD data management proxies. Possible integration is depicted in Figure 1.5.

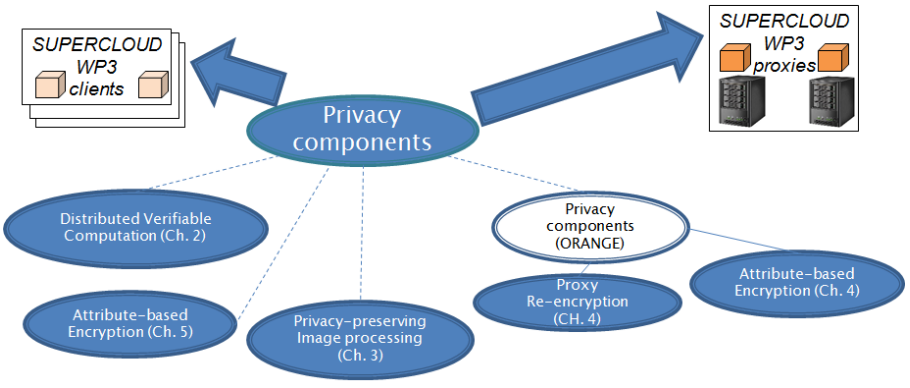


Figure 1.5: Privacy-preserving components integration vector and its fit within the SUPERCLOUD data management architecture. Blue components are described in this deliverable in respective chapters. Full lines represent single-partner integration vectors, dashed lines represent integration vectors across two or more partners.

Part I

State-machine replication

Chapter 2 State-Machine Replication with Hyperledger Blockchain Fabric

2.1 Overview

State-machine replication [278] is a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas. A replicated service maintains some state and clients invoke operations that transform the state and generate outputs.
In cloud computing, state-machine replication is widely used for replicating critical services such as configuration management or metadata replication (e.g., [181]) tolerating primarilycrash faults of clients and replicas. In SUPERCLOUD, replicas are to be placed across potentially untrusted clouds and the state-machine replication needs to be resilient to arbitrary (also called Byzantine [212]) faults of clients and replicas.
These SUPERCLOUD needs from state-machine replication are largely in line with the emerging blockchain technology [309]. A blockhain can be modeled as a replicated state-machine and emulates a "trusted" computing service through a distributed protocol, run by nodes connected over the Internet. The nodes share the common goal of running the service but do not necessarily trust each other for more. In a "permissionless" blockchain such as the one underlying the Bitcoin cryptocurrency, anyone can operate a node and participate through spending CPU cycles and demonstrating a "proof-of-work." On the other hand, blockchains in the "permissioned" model control who participates in validation and in the protocol; these nodes typically have established identities and form a consortium. A report of Swanson compares the two models [292].
To leverage synergies and similarities with blockchain, SUPERCLOUD state-machine replication will be based around Hyperledger fabric (`github.com/hyperledger/fabric`) open-source blockchain project that will serve as an envelope for state-machine replication in SUPERCLOUD. Hyperledger fabric is a part of the Hyperledger Project (`www.hyperledger.org`), a major open source blockchain effort developed in collaboration across several industries, with the goal to effort to create an enterprise-grade, open-source distributed ledger framework and code base. It aims to advance blockchain technology by identifying and realizing a cross-industry open standard platform for distributed ledgers, which can transform the way business transactions are conducted globally. Established as a project of the Linux Foundation in early 2016, the Hyperledger Project currently has already more than 90 members, with IBM being one of the premier and founding members.
SUPERCLOUD project tightly collaborates with Hyperledger project, notably through contributions from IBM. This colaboration enhances the visibility of the SUPERCLOUD project and ensures that results of the project, and WP3 in particular, are contributed to a major open-source projects.
In this Chapter we briefly explain the high-level details of Hyperledger fabric. In Chapters that follow, we explain novel state-machine replication components and techniques developed in SUPERCLOUD for which Hyperledger fabric serves as an integration vector (see Chapter 1.3).

2.2 Hyperledger Fabric

Hyperledger Fabric (`github.com/hyperledger/fabric`) is an implementation of a distributed ledger platform for running smart contracts, leveraging familiar and proven technologies, with a modular

architecture allowing pluggable implementations of various functions. It is one of multiple projects currently in incubation under the Hyperledger Project. A developer-preview of the Hyperledger Fabric (called "v0.5-developer-preview") has been released in June 2016 (`github.com/hyperledger/fabric/wiki/Fabric-Releases`).
The distributed ledger protocol of the fabric is run by peers. The fabric distinguishes between two kinds of peers: A validating peer is a node on the network responsible for running consensus, validating transactions, and maintaining the ledger. On the other hand, a non-validating peer is a node that functions as a proxy to connect clients (issuing transactions) to validating peers. A non-validating peer does not execute transactions but it may verify them.
Some key features of the current fabric release are:

- A permissioned blockchain with immediate finality;
- Runs arbitrary smart contracts (called chaincode) implemented in Go (`golang.org`):
  - User-defined chaincode is encapsulated in a Docker container;
  - System chaincode runs in the same process as the peer;
- Consensus protocol is pluggable, currently an implementation of Byzantine fault-tolerant consensus using the PBFT protocol [96] is supported, a prototype of SIEVE (see Chapter 3) to address non-deterministic chaincode is available, and a protocol stub (named NOOPS) serves for development on a single node;
- Security support through certificate authorities (CAs) for TLS certificates, enrollment certificates, and transaction certificates;
- Persistent state using a key-value store interface, backed by RocksDB (`rocksdb.org`);
- An event framework that supports pre-defined and custom events;
- A client SDK (Node.js) to interface with the fabric;
- Support for basic REST APIs and CLIs.

Support for non-validating peers is minimal in the developer preview release.

2.3 Architecture

The validating peers run a BFT consensus protocol for executing a replicated state machine that accepts three types of transactions as operations:

Deploy transaction: Takes a chaincode (representing a smart contract) written in Go as a parameter; the chaincode is installed on the peers and ready to be invoked.

Invoke transaction: Invokes a transaction of a particular chaincode that has been installed earlier through a deploy transaction; the arguments are specific to the type of transaction; the chaincode executes the transaction, may read and write entries in its state accordingly, and indicates whether it succeeded or failed.

Query transaction: Returns an entry of the state directly from reading the peer's persistent state; this may not ensure linearizability.

Each chaincode may define its own persistent entries in the state. The blockchain's hash chain is computed over the executed transactions and the resulting persistent state.
Validation of transactions occurs through the replicated execution of the chaincode and given the fault assumption underlying BFT consensus, i.e., that among the n validating peers at most $f < n/3$ may "lie" and behave arbitrarily, but all others execute the chaincode correctly. When executed on top of PBFT consensus, it is important that chaincode transactions are deterministic, otherwise the state of the peers might diverge. A modular solution to filter out non-deterministic transactions that are demonstrably diverging is available and has been implemented in the SIEVE protocol [85].

Membership among the validating nodes running BFT consensus is currently static and the setup requires manual intervention. Support for dynamically changing the set of nodes running consensus is planned for a future version.
As the fabric implements a permissioned ledger, it contains a security infrastructure for authentication and authorization. It supports enrollment and transaction authorization through public-key certificates, and confidentiality for chaincode realized through in-band encryption.
More precisely, for connecting to the network every peer needs to obtain an enrollment certificate from an enrollment CA that is part of the membership services. It authorizes a peer to connect to the network and to acquire transaction certificates, which are needed to submit transactions. Transaction certificates are issued by a transaction CA and support pseudonymous authorization for the peers submitting transactions, in the sense that multiple transaction certificates issued to the same peer (that is, to the same enrollment certificate) cannot be linked with each other.
Confidentiality for chaincodes and state is provided through symmetric-key encryption of transactions and states with a blockchain-specific key that is available to all peers with an enrollment certificate for the blockchain. Extending the encryption mechanisms towards more fine-grained confidentiality for transactions and state entries is planned for a future version.

2.4 Discussion

Consensus protocols for blockchain are currently being debated very actively, in research [156, 309] but also by fintech startup companies (e.g., `tendermint.com`, `kadena.io`). The fabric's design uses a modular notion of consensus, which is aligned with the well-established concept of consensus in distributed computing. This ensures that the blockchain-related features of the fabric can be developed independently of the specific consensus protocol. The PBFT protocol [96] has been implemented as the first one in the fabric because of its prominence: it benefits from the experience of almost 20 years of systems-level research on Byzantine consensus, is closely related to well-known protocols like viewstamped replication and Paxos [218], has been analyzed in many environments [113, 37], and is described in textbooks [81].

2.5 Conclusion

The Hyperledger Fabric is a permissioned blockchain platform aimed at business use. It is open-source and based on standards, runs user-defined smart contracts, supports strong security and identity features, and uses a modular architecture with pluggable consensus protocols.
The fabric is currently evolving and being actively developed under the governance of the Hyperledger Project. More information about the fabric is available online at
`github.com/hyperledger/fabric/blob/master/docs/protocol-spec.md`

Chapter 3 Non-deterministic Byzantine Fault-Tolerant State-Machine Replication

Service replication distributes an application over many processes for tolerating faults, attacks, and misbehavior among a subset of the processes. With the recent interest in multi-cloud architectures such as the one considered in SUPERCLOUD, as well as blockchain technologies (such as Hyperledger - see Chapter 2), distributed execution of one logical application over multiple administrative domains has become a prominent topic. The established state-machine replication paradigm inherently requires the application to be deterministic. However, applications developed in high-level languages are seldom deterministic and require an effort from the application developer to ensure determinism of the application.
In this chapter we consider and proposer new techniques for non-deterministic Byzantine fault-tolerant state-machine replication. We distinguish three models for dealing with non-determinism in replicated services, where some processes are subject to Byzantine faults: first, the modular case that does not require any changes to the potentially non-deterministic application (and neither access to its internal data); second, master-slave solutions, where ties are broken by a leader and the other processes validate the choices of the leader; and finally, applications that use cryptography and secret keys. Cryptographic operations and secrets must be treated specially because they require strong randomness to satisfy their goals.
The chapter also introduces two new protocols. First, Protocol Sieve uses the modular approach and filters out non-deterministic operations in an application. It ensures that all correct processes produce the same outputs and that their internal states do not diverge. A second protocol, called Mastercrypt, implements cryptographically secure randomness generation with a verifiable random function and is appropriate for most situations in which cryptographic secrets are involved. All protocols are described in a generic way and do not assume a particular implementation of the underlying consensus primitive.

3.1 Introduction

State-machine replication is an established way to enhance the resilience of a client-server application [278]. It works by executing the service on multiple independent components that will not exhibit correlated failures. We consider the approach of Byzantine fault-tolerance (BFT), where a group of processes connected only by an unreliable network executes an application [257]. The processes use a protocol for consensus or atomic broadcast to agree on a common sequence of operations to execute. If all processes start from the same initial state, if all operations that modify the state are deterministic, and if all processes execute the same sequence of operations, then the states of the correct processes will remain the same. (This is also called active replication [102].) A client executes an operation on the service by sending the operation to all processes; it obtains the correct outcome based on comparing the responses that it receives, for example, by a relative majority among the answers or from a sufficiently large set of equal responses. Tolerating Byzantine faults means that the clients obtain correct outputs as long as a qualified majority of the processes is correct, even if the faulty processes behave in arbitrary and adversarial ways.
Traditionally state-machine replication requires the application to be deterministic. But many applications contain implicit or explicit non-determinism: in multi-threaded applications, the scheduler

may influence the execution, input/output operations might yield different results across the processes, probabilistic algorithms may access a random-number generator, and some cryptographic operations are inherently not deterministic.
For practical use of state-machine replication, for deploying custom developed applications, ensuring deterministic operations is crucial since even the smallest divergence among the outputs of different participants introduces inconsistency.
This chapter presents a general treatment of non-determinism in the context of BFT replication and introduces a distinction among different models to tackle the problem of non-determinism. For example, applications involving cryptography and secret encryption keys should be treated differently from those that access randomness for other goals. We also distinguish whether the replication mechanism has access to the application's source code and may modify it.
We also introduce two novel protocols. The first, called Sieve, replicates non-deterministic programs using in a modular way, where we treat the application as a black box and cannot change it. We target workloads that are usually deterministic, but which may occasionally yield diverging outputs. The protocol initially executes all operations speculatively and then compares the outputs across the processes. If the protocol detects a minor divergence among a small number of processes, then we sieve out the diverging values; if a divergence among too many processes occurs, we sieve out the operation from sequence. Furthermore, the protocol can use any underlying consensus primitive to agree on an ordering. The second new protocol, Mastercrypt, provides master-slave replication with cryptographic security from verifiable random functions. It addresses situations that require strong, cryptographically secure randomness, but where the faulty processes may leak their secrets.

3.2 Definitions

3.2.1 System model

We consider a distributed system of processes that communicate with each other and provide a common service in a fault-tolerant way. Using the paradigm of service replication [278], requests to the service are broadcast among the processes, such that the processes execute all requests in the same order. The clients accessing the service are not modeled here. We denote the set of processes by $\mathsf{CP}$and let $n = |\mathsf{CP}|$. A process may be faulty, by crashing or by exhibiting Byzantine faults; the latter means they may deviate arbitrarily from their specification. Non-faulty processes are called correct. Up to f processes may be faulty and we assume that $n > 3f$. The setup is also called a Byzantine fault-tolerant (BFT) service replication system or simply a BFT system.
We present protocols in a modular way using an event-based notation [81]. A process is specified through its interface, consumes input events, and generates output events. Every two processes can send messages to each other using an authenticated point-to-point communication primitive. When a message arrives, the receiver learns also which process has sent the message. The primitive guarantees message integrity, i.e., when a message m is received by a correct process with indicated sender p_s, and p_s is correct, then p_s previously sent m.
The system is partially synchronous [141] in the sense that there is no a priori bound on message delays and the processes have no synchronized clocks, as in an asynchronous system. However, there is a time (not known to the processes) after which the system is stable in the sense that message delays and processing times are bounded. In other words, the system is eventually synchronous. This model represents a broadly accepted network model and covers a wide range of real-world situations.

3.2.2 Broadcast and state-machine replication

Atomic broadcast. Suppose n processes participate in a broadcast primitive. Every process may broadcast a request or message m to the others. When a request has been agreed, it is delivered. Atomic broadcast also solves the consensus problem [170, 81]. We use a variant that delivers only messages satisfying a given external validity condition [83].

More precisely, Byzantine atomic broadcast with external validity (abv) is defined with a validation predicate $V()$ and uses two events: *abv-broadcast*(m), to broadcast a message m to all processes, and *abv-deliver*(p, m), which delivers a message m broadcast by process p.
Predicate $V()$ validates messages. It can be computed locally by every process and ensures that a correct process only delivers messages that satisfy $V()$. More precisely, $V()$ must guarantee that when two correct processes p and q have both delivered the same sequence of messages up to some point, then p obtains $V(m) = \text{TRUE}$ for any message m if and only if q also determines that $V(m) = \text{TRUE}$. The standard properties of Byzantine atomic broadcast [81] (validity, no duplication, integrity, agreement, and total order) are extended by:

External validity: When a correct process delivers some message m, then $V(m) = \text{TRUE}$.

In practice it may occur that not all processes agree in the above sense on the validity of a message. For instance, some correct process may conclude $V(m) = \text{TRUE}$ while others find that $V(m) = \text{FALSE}$. For this case it is useful to reason with the following relaxation:

Weak external validity: When a correct process delivers some message m, then at least one correct process has determined that $V(m) = \text{TRUE}$ at some time between when m was broadcast and when it was delivered.

Every protocol for Byzantine atomic broadcast with external validity of which we are aware either ensures this weaker notion or can easily be changed to satisfy it.

State machine replication. Atomic broadcast is the main tool to implement state-machine replication (SMR), which executes a service on multiple processes for tolerating process faults. Throughout this work we assume that many operation requests are generated concurrently by all processes; in other words, there is request contention.
A state machine consists of variables and operations that transform its state and may produce some output. Traditionally, operations are deterministic. The state machine functionality is defined by *execute*(), a function that takes a state $s \in \mathcal{S}$, initially s_0, and operation $o \in \mathcal{O}$ as input, and outputs a successor state s' and a response or output value r, such that *execute*$(s, o) \rightarrow (s', r)$.
A replicated state-machine is defined by two events: an input event *rsm-execute*(*operation*) that a process uses to invoke the execution of an operation o of the state machine; and an output event *rsm-output*(o, s, r), which is produced by the state machine. The output indicates the operation has been executed and carries the resulting state s and response r. We assume here that an operation o includes both the name of the operation to be executed and any relevant parameters.
More formally, a replicated state machine (rsm) receives requests that the state machine executes the operation o, in the form of *rsm-execute*(o) events; it produces *rsm-output*(o, s, r) events, to indicate that the state machine has executed an operation o, resulting in new state s, and producing response r. It is defined using standard properties [81], ensuring agreement on the executed sequence of operations among all correct processes; correctness in the sense that when a correct process has executed a sequence of operations $o_1, \ldots, o_k$, then the sequences of output states $s_1, \ldots, s_k$ and responses $r_1, \ldots, r_k$ satisfies $(s_i, r_i) = \textit{execute}(s_{i-1}, o_i)$ for $i = 1, \ldots, k$; and finally, termination.
The standard implementation of a replicated state machine relies on an atomic broadcast protocol to disseminate the requests to all processes [278, 170].

3.2.3 Leader election

Implementations of atomic broadcast need to make some synchrony assumptions or employ randomization [148]. A very weak timing assumption that is also available in many practical implementations is an eventual leader-detector oracle [101, 170].
We define an eventual leader-detector primitive, denoted Ω, for a system with Byzantine processes. It informs the processes about one correct process that can serve as a leader, so that the protocol can

progress. When faults are limited to crashes, such a leader detector can be implemented from a failure detector [101], a primitive that, in practice, exploits timeouts and low-level point-to-point messages to determine whether a remote process is alive or has crashed.
With processes acting in arbitrary ways, though, one cannot rely on the timeliness of simple responses for detecting Byzantine faults. One needs another way to determine remotely whether a process is faulty or performs correctly as a leader. Detecting misbehavior in this model depends inherently on the specific protocol being executed [138]. We use the approach of "trust, but verify," where the processes monitor the leader for correct behavior. More precisely, a leader is chosen arbitrarily, but ensuring a fair distribution among all processes (in fact, it is only needed that a correct process is chosen at least with constant probability on average, over all leader changes). Once elected, the chosen leader process gets a chance to perform well. The other processes monitor its actions. Should the leader not have achieved the desired goal after some time, they complain against it, and initiate a switch to a new leader.
This notion of "performance" depends on the specific algorithm executed by the processes, which relies on the output from the leader-detection module. Therefore, eventual leader election with Byzantine processes is not an isolated low-level abstraction, as with crash-stop processes, but requires some input from the higher-level algorithm. The Ω-*complain*(p) event allows to express this. Every process may complain against the current leader p by triggering this event.
Formally, a Byzantine leader detector (Ω) is defined with an output Ω-*trust*(p), designating process p to be trusted as leader, and an input event Ω-*complain*(p) that receives a complaint about the performance of leader process p. Its formal properties [81] ensure that eventually, every correct process trusts some correct process; that when more than f correct processes that trust some process p complain about p, then every correct process eventually trusts a different process than p. Moreover, a correct process q does not trust a new leader unless at least one correct process has complained against the leader which q trusted before, and that eventually no two correct processes trust different processes.
It is possible to lift the output from the Byzantine leader detector to an epoch-change primitive, which outputs not only the identity of a leader but also an increasing epoch number. This abstraction divides time into a series of epochs at every participating process, where epochs are identified by numbers. The numbers of the epochs started by one particular process increase monotonically (but they do not have to form a complete sequence). Moreover, the primitive also assigns a leader to every epoch, such that any two correct processes in the same epoch receive the same leader. The mechanism for processes to complain about the leader is the same as for Ω.
More precisely, Byzantine epoch-change (Ψ) outputs events of the form Ψ-*start-epoch*(e, p), which indicate that epoch with number e and leader p starts; it also receives Ψ-*complain*(e, p) events similar to Ω. Its formal properties appears in the literature [81].
When an epoch-change abstraction is initialized, it is assumed that a default epoch with number 0 and a leader p_0 has been started at all correct processes. All "practical" BFT systems in the eventual-synchrony model starting from PBFT [96] implicitly contain an implementation of Byzantine epoch-change; this notion was described explicitly by Cachin et al. [81, Chap. 5].

3.3 Modular protocol

In this section we discuss the modular execution of replicated non-deterministic programs. Here the program is given as a black box, it cannot be changed, and the BFT system cannot access its internal data structures. Very informally speaking, if some processes arrive at a different output during execution than "most" others, then the output of the disagreeing processes is discarded. Instead they should "adopt" the output of the others, e.g., by asking them for the agreed-on state and response. When the outputs of "too many" processes disagree, the correct output may not be clear; the operation is then ignored (or, as an optimization, quarantined as non-deterministic) and the state rolled back. In this modular solution any application can be replicated without change; the application developers may not even be aware of potential non-determinism. On the other hand, the modular protocol

requires that most operations are deterministic and produce almost always the same outputs at all processes; it would not work for replicating probabilistic functions.
More precisely, a non-deterministic state machine may output different states and responses for the same operation, which are due to probabilistic choices or other non-repeatable effects. Hence we assume that *execute* is a relation and not a deterministic function, that is, repeated invocations of the same operation with the same input may yield different outputs and responses. This means that the standard approach of state-machine replication based directly on atomic broadcast fails.
There are two ways for modular black-box replication of non-deterministic applications in a BFT system:

Order-then-execute: Applying the SMR principle directly, the operations are first ordered by atomic broadcast. Whenever a process delivers an operation according to the total order, it executes the operation. It does not output the response, however, before checking with enough others that they all arrive at the same outputs. To this end, every process atomically broadcasts its outputs (or a hash of the outputs) and waits for receiving a given number (up to $n - f$) of outputs from distinct processes. Then the process applies a fixed decision function to the atomically delivered outputs, and it determines the successor state and the response.

This approach ensures consistency due to its conceptual simplicity but is not very efficient in typical situations, where atomic broadcast forms the bottleneck. In particular, in atomic broadcast with external validity, a process can only participate in the ordering of the next operation when it has determined the outputs of the previous one. This eliminates potential gains from pipelining and increases the overall latency.

Execute-then-order: Here the steps are inverted and the operations are executed speculatively before the system commits their order. As in other practical protocols, this solution uses the heuristic assumption that there is a designated leader which is usually correct. Thus, every process sends its operations to the leader and the leader orders them. It asks all processes to execute the operations speculatively in this order, the processes send (a hash of) their outputs to the leader, and the leader determines a unique output. Note that this value is still speculative because the leader might fail or there might be multiple leaders acting concurrently. The leader then tries to obtain a confirmation of its speculative order by atomically broadcasting the chosen output. Once every process obtains this output from atomic broadcast, it commits the speculative state and outputs the response.

In rare cases when a leader is replaced, some processes may have speculated wrongly and executed other operations than those determined through atomic broadcast. Due to non-determinism in the execution a process may also have obtained a different speculative state and response than what the leader has obtained and broadcast. This implies that the leader must either send the state (or state delta) and the response resulting from the operation though atomic broadcast, or that a process has a different way to recover the decided state from other processes.

In the following we describe Protocol Sieve, which adopts the approach of execute-then-order with speculative execution.

Protocol Sieve. Protocol Sieve runs a Byzantine atomic broadcast with weak external validity (abv) and uses a sieve-leader to coordinate the execution of non-deterministic operations. The leader is elected through a Byzantine epoch-change abstraction, as defined in Section 3.2.3, which outputs epoch/leader tuples with monotonically increasing epoch numbers. For the Sieve protocol these epochs are called configurations, and Sieve progresses through a series of them, each with its own sieve-leader. The processes send all operations to the service through the leader of the current configuration, using an INVOKE message. The current leader then initiates that all processes execute the operation speculatively; subsequently the processes agree on an output from the operation and thereby commit

the operation. As described here, Sieve executes one operation at a time, although it is possible to greatly increase the throughput using the standard method of batching multiple operations together. The leader sends an EXECUTE message to all processes with the operation o. In turn, every process executes o speculatively on its current state s, obtains the speculative next state t and the speculative response r, signs those values, and sends a hash and the signature back to the leader in an APPROVE message.

The leader receives $2f + 1$ APPROVE messages from distinct processes. If the leader observes at least $f + 1$ approvals for the same speculative output, then it confirms the operation and proceeds to committing and executing it. Otherwise, the leader concludes that the operation is aborted because of diverging outputs. There must be $f + 1$ equal outputs for confirming o, in order to ensure that every process will eventually learn the correct output, see below.

The leader then *abv-broadcasts* an ORDER message, containing the operation, the speculative output (t, r) for a confirmed operation or an indication that it aborted, and for validation the set of APPROVE messages that justify the decision whether to confirm or abort. During atomic broadcast, the external validity check by the processes will verify this justification.

As soon as an ORDER message with operation o is abv-delivered to a process in Sieve, o is committed. If o is confirmed, the process adopts the output decided by the leader. Note this may differ from the speculative output computed by the process. Protocol Sieve therefore includes the next state t and the response r in the ORDER message. In practice, however, one might not send t, but state deltas, or even only the hash value of t while relying on a different way to recover the confirmed state. Indeed, since $f + 1$ processes have approved any confirmed output, a process with a wrong speculative output is sure to reach at least one of them for obtaining the confirmed output later.

In case the leader abv-broadcasted an ORDER message with the decision to abort the current operation because of the diverging outputs (i.e., no $f + 1$ identical hashes in $2f + 1$ APPROVE messages), the process simply ignores the current request and speculative state. As an optimization, processes may quarantine the current request and flag it as non-deterministic.

As described so far, the protocol is open to a denial-of-service attack by multiple faulty processes disguising as sieve-leaders and executing different operations. Note that the epoch-change abstraction, in periods of asynchrony, will not ensure that any two correct processes agree on the leader, as some processes might skip configurations. Therefore Sieve also orders the configuration and leader changes using consensus (with the abv primitive).

To this effect, whenever a process receives a *start-epoch* event with itself as leader, the process *abv-broadcasts* a NEW-SIEVE-CONFIG message, announcing itself as the leader. The validation predicate for broadcast verifies that the leader announcement concerns a configuration that is not newer than the most recently started epoch at the validating process, and that the process itself endorses the same next leader. Every process then starts the new configuration when the NEW-SIEVE-CONFIG message is *abv-delivered*. If there was a speculatively executed operation, it is aborted and its output discarded.

The design of Sieve prevents uncoordinated speculative request execution, which may cause contention among requests from different self-proclaimed leaders and can prevent liveness easily. Naturally, a faulty leader may also violate liveness, but this is not different from other leader-based BFT protocols.

The details of Protocol Sieve are shown in Algorithms 1–2. The pseudocode assumes that all point-to-point messages among correct processes are authenticated, cannot be forged or altered, and respect FIFO order. The invoked operations are unique across all processes and *self* denotes the identifier of the executing process. Not shown in the pseudocode is a periodic concurrent check for leader progress. The process determines the age of every $o \in \mathcal{I}$ since it has been invoked and added to $\mathcal{I}$; if there are "old" operations in $\mathcal{I}$, then the process invokes Ψ-*complain*(*leader*).

The following two optimizations for Sieve are described in the full version [85]: First, when run in practice, every process directly executes operations and does not include the potentially large state in ORDER messages. If a rollback operation exists to complement *execute*, a process that has computed a diverging state can roll the operation back and obtain the state from other processes. Second, when the well-known PBFT protocol [96] implements *abv-broadcast*, then the leader information and Byzantine

Algorithm 1 Protocol Sieve

State

$\mathcal{I}$: set of invoked operations at every process — $B[p]$, for $p \in \mathsf{CP}$: buffer at sieve-leader
config: sieve-config number — *leader*: sieve-leader, initially p_0
next-epoch: next sieve-config, initially $\perp$ — *next-leader*: next sieve-leader, initially $\perp$
s: current state, initially s_0 — *cur*: current operation, initially $\perp$
t: speculative state, initially $\perp$ — r: speculative response, initially $\perp$

upon invocation *rsm-execute*(o) **do**
  $\mathcal{I} \leftarrow \mathcal{I} \cup \{o\}$
  send msg. [INVOKE, *config*, o] over point-to-point link to *leader*

upon recv. msg. [INVOKE, c, o] from p **such that** $B[p] = \perp$ **and** $c =$ *config* **and** *leader* $=$ *self* **do**
  $B[p] \leftarrow o$ // buffer only the latest operation from each process

upon exists p that $B[p] \neq \perp$ **such that** *cur* $= \perp$ **and** *leader* $=$ *self* **do**
  cur $\leftarrow B[p]$
  send [EXECUTE, *config*, *cur*] over point-to-point links to all processes

upon recv. msg. [EXECUTE, c, o] from p **such that** $p =$ *leader* **and** $c =$ *config* **and** $t = \perp$ **do**
  $(t, r) \leftarrow$ *execute*(s, o)
  $\sigma \leftarrow sign_{self}$(SPECULATE$\|$*config*$\|hash(t\|r)$)
  send msg. [APPROVE, *config*, o, $hash(t\|r), \sigma$] to *leader*

upon recv. $2f + 1$ msgs. [APPROVE, c_p, o_p, h_p, σ_p], each from a distinct process p, **such that**
    $c_p =$ *config* **and** $op_p =$ *cur* **and** $verify_p(\sigma_p,$ SPECULATE$\|$*config*$\|h_p)$ **and** *leader* $=$ *self* **do**
  if there is a set $\mathcal{E}$ of $f + 1$ received APPROVE msgs. whose h_p value is equal to $hash(t\|r)$ **then**
    abv-broadcast([ORDER, CONFIRM, *config*, *cur*, $t, r, \mathcal{E}$])
  else
    let $\mathcal{U}$ be the set of $2f + 1$ received APPROVE msgs.
    abv-broadcast([ORDER, ABORT, *config*, *cur*, $\perp, \perp, \mathcal{U}$])

upon *abv-deliver*(p, [ORDER, *decision*, $c, o, t_c, r_c, \cdot$]) **such that** $c =$ *config* **do** // commit o
  if *leader* $=$ *self* **then**
    $B[p] \leftarrow \perp$
    cur $\leftarrow \perp$
  if $o \in \mathcal{I}$ **then**
    $\mathcal{I} \leftarrow \mathcal{I} \setminus \{o\}$
  if *decision* $=$ CONFIRM **then**
    $s \leftarrow t_c$ // adopt the agreed-on state and response, needed if $(t_c, r_c) \neq (t, r)$
    rsm-output(o, s, r_c)
  $t \leftarrow \perp$

upon Ψ-*start-epoch*(e, p) **do**
  (*next-epoch*, *next-leader*) $\leftarrow (e, p)$
  if $p =$ *self* $\wedge\ e >$ *config* **then**
    abv-broadcast([NEW-SIEVE-CONFIG, e, *self*])

upon *abv-deliver*(p, [NEW-SIEVE-CONFIG, c, p]) **do**
  (*config*, *leader*) $\leftarrow (c, p)$
  $t \leftarrow \perp$

Algorithm 2 Validation predicate $V()$ for Byzantine atomic broadcast used inside Algorithm Sieve

upon invocation $V(m)$ **do**
  if $m = [\text{ORDER}, \text{DECISION}, c, o, \mathcal{M}]$ **then**
    if $\mathcal{M}$ is a set of $f+1$ msgs. of the form $[\text{APPROVE}, c_p, o_p, h_p, \sigma_p]$ **such that**
      $c_p = config$ **and** $o_p = o$ **and** $verify_p(\sigma_p, \text{SPECULATE}\|c_p\|h_p) = \text{TRUE}$ **and**
      all h_p values in $\mathcal{M}$ are equal **then**
    return TRUE
  else if $m = [\text{ORDER}, \text{ABORT}, c, o, \mathcal{M}]$ **then**
    if $\mathcal{M}$ is a set of $2f+1$ msgs. of the form $[\text{APPROVE}, c_p, o_p, h_p, \sigma_p]$ **such that**
      $c_p = config$ **and** $o_p = o$ **and** $verify_p(\sigma_p, \text{SPECULATE}\|c_p\|h_p) = \text{TRUE}$ **and**
      no $f+1$ of the h_p values in $\mathcal{M}$ are equal **then**
    return TRUE
  else if $m = [\text{NEW-SIEVE-CONFIG}, c, p]$ **then**
    if $c \leq next\text{-}epoch$ **and** $p = next\text{-}leader$ **then**
      return TRUE
  return FALSE

epoch-change mechanism can be directly obtained from PBFT. This simplifies the description of Sieve but breaks modularity.

Theorem 1. *Protocol Sieve implements a replicated state machine allowing a non-deterministic functionality* execute()*, except that demonstrably non-deterministic operations may be filtered out and not executed.*

To see why this holds, we consider first the agreement condition of a replicated state machine: this follows directly from the protocol and from the abv primitive. Every *rsm-output* event is immediately preceded by an *abv-delivered* ORDER message, which is the same for all correct processes due to agreement of abv. Since all correct processes react to it deterministically, their outputs are the same. For the correctness property, note that the outputs (s_i, r_i) (state and response) resulting from an operation o must have been confirmed by the protocol and therefore the values were included in an APPROVE message from at least one correct process. This process computed the values such that they satisfy $(s_i, r_i) = execute(s_{i-1}, o)$ according to the protocol for handling an EXECUTE message. On the other hand, no correct process outputs anything for committed operations that were aborted, this is permitted by the exception in the theorem statement. Moreover, only operations are filtered out for which distinct correct processes computed diverging outputs, as ensured by the sieve-leader when it determines whether the operation is confirmed or aborted. In order to abort, no set of $f+1$ processes must have computed the same outputs among the $2f+1$ processes sending the APPROVE messages. Hence, at least two among every set of $f+1$ correct processes arrived at diverging outputs.
Termination is only required for deterministic operations, they must terminate despite faulty processes that approve wrong outputs. The protocol ensures this through the condition that at least $f+1$ among the $2f+1$ APPROVE messages received by the sieve-leader are equal. The faulty processes, of which there are at most f, cannot cause an abort through this. But every ORDER message is eventually *abv-delivered* and every confirmed operation is eventually executed and generates an output.

Discussion. Non-deterministic operations have not often been discussed in the context of BFT systems. The literature commonly assumes that deterministic behavior can be imposed on an application or postulates to change the application code for isolating non-determinism. In practice, however, it is often not possible.
Liskov [218] sketches an approach to deal with non-determinism in PBFT which is similar to Sieve in the sense that it treats the application code modularly and uses execute-then-order. This proposal is restricted to the particular structure of PBFT, however, and does not consider the notion of external validity for abv broadcast.

For applications on multi-core servers, the Eve system [192] also executes operation groups speculatively across processes and detects diverging states during a subsequent verification stage. In case of divergence, the processes must roll back the operations. The approach taken in Eve resembles that of Sieve, but there are notable differences. Specifically, the primary application of Eve continues to assume deterministic operations, and non-determinism may only result from concurrency during parallel execution of requests. Furthermore, this work uses a particular agreement protocol based on PBFT and not a generic abv broadcast primitive.
It should be noted that Sieve not only works with Byzantine atomic broadcast in the model of eventual synchrony, but can equally well be run over randomized Byzantine consensus [83, 236].

3.4 Master-slave protocol

By adopting the master-slave model one can support a broader range of non-deterministic application behavior compared to the modular protocol. This design generally requires source-code access and modifications to the program implementing the functionality. In a master-slave protocol for non-deterministic execution, one process is designated as master. The master executes every operation first and records all non-deterministic choices. All other processes act as slaves and follow the same choices. To cope with a potentially Byzantine master, the slaves must be given means to verify that the choices made by the master are plausible. The master-slave solution presented here follows primary-backup replication [77], which is well-known to handle non-deterministic operations. For instance, if the application accesses a pseudorandom number generator, only the master obtains the random bits from the generator and the slaves adopt the bits chosen by the master. This protocol does not work for functionalities involving cryptography, however, where master-slave replication typically falls short of achieving the desired goals. Instead a cryptographically secure protocol should be used; they are the subject of Section 3.5.

Non-deterministic execution with evidence. As introduced in Section 3.3, the *execute* operation of a non-deterministic state machine is a relation. Different output values are possible and represent acceptable outcomes. We augment the output of an operation execution by adding evidence for justifying the resulting state and response. The slave processes may then replay the choices of the master and accept its output.
More formally, we now extend *execute* to *nondet-execute* as follows:

$$\textit{nondet-execute}(s, o) \rightarrow (s', r, \rho).$$

Its parameters s, o, s', and r are the same as for *execute*; additionally, the function also outputs evidence ρ. Evidence enables the slave processes to execute the operation by themselves and obtain the same output as the master, or perhaps only to validate the output generated by another execution. For this task there is a function

$$\textit{verify-execution}(s, o, s', r, \rho) \rightarrow \{\text{FALSE}, \text{TRUE}\}$$

that outputs TRUE if and only if the set of possible outputs from $\textit{nondet-execute}(s, o)$ contains (s', r, ρ). For completeness we require that for every s and o, when $(s', r, \rho) \leftarrow \textit{nondet-execute}(s, o)$, it always holds $\textit{verify-execute}(s, o, s', r, \rho) = \text{TRUE}$.
As a basic verification method, a slave could rerun the computation of the master. Extensions to use cryptographic verifiable computation [310] are possible. Note that we consider randomized algorithms to be a special case of non-deterministic ones. The evidence for executing a randomized algorithm might simply consist of the random coin flips made during the execution.

Replication protocol. Implementing a replicated state machine with non-deterministic operations using master-slave replication does not require an extra round of messages to be exchanged, as in

Protocol Sieve. It suffices that the master is chosen by a Byzantine epoch-change abstraction and that the master broadcasts every operation together with the corresponding evidence.
More precisely, the processes operate on top of an underlying broadcast primitive abv and a Byzantine epoch-change abstraction Ψ. Whenever a process receives a *start-epoch* event with itself as leader from Ψ, the process considers itself to be the master for the epoch and *abv-broadcasts* a message that announces itself as the master for the epoch. The epochs evolve analogously to the configurations in Sieve, with the same mechanism to approve changes of the master in the validation predicate of atomic broadcast. Similarly, non-master processes send their operations to the master of the current epoch for ordering and execution.
For every invoked operation o, the master computes $(s', r, \rho) \leftarrow$ *nondet-execute*(s, o) and *abv-broadcasts* an ORDER message containing the current epoch c and parameters o, s', r, and ρ. The validation predicate of atomic broadcast for ORDER messages verifies that the message concerns the current epoch and that *verify-execution*$(s, o, s', r, \rho) =$ TRUE using the current state s of the process. Once an ORDER message is *abv-delivered*, a process adopts the response and output state from the message as its own.

Discussion. The master-slave protocol is inspired by primary-backup replication [77], and for the concrete scenario of a BFT system, it was first described by Castro, Rodrigues, and Liskov in BASE [97]. The protocol of BASE addresses only the particular context of PBFT, however, and not a generic atomic broadcast primitive. As mentioned before, the master-slave protocol requires changes to the application for extracting the evidence that will convince the slave processes that choices made by the master are valid.

3.5 Cryptographically secure protocols

Security functions implemented with cryptography are more important today than ever. Replicating an application that involves a cryptographic secret, however, requires a careful consideration of the attack model. If the BFT system should tolerate that f processes become faulty in arbitrary ways, it must be assumed that their secrets leak to the adversary against whom the cryptographic scheme is employed.
Service-level secret keys must be protected and should never leak to an individual process. Two solutions have been explored to address this issue. One could delegate this responsibility to a third party, such as a centralized service or a secure hardware module at every process. However, this contradicts the main motivation behind replication: to eliminate central control points. Alternatively one may use distributed cryptography [131], share the keys among the processes so that no coalition of up to f among them learns anything, and perform the cryptographic operations under distributed control. This model was pioneered by Reiter and Birman [269] and exploited, for instance, by SINTRA [80, 84] or COCA [322].
In this section we introduce a novel protocol, called Mastercrypt, for integrating non-deterministic cryptographic operations in a BFT system, based on the master-slave paradigm and using verifiable random functions to generate pseudorandom bits. This randomness is unpredictable and cannot be biased by a Byzantine process. In the full version [85] we furthermore review a protocol based on the well-known idea of using distributed cryptography, as discussed above. Both schemes adopt the master-slave replication protocol from the previous section.

Randomness from verifiable random functions. A verifiable random function (VRF) [234] resembles a pseudorandom function but additionally permits anyone to verify non-interactively that the choice of random bits occurred correctly. The function therefore guarantees correctness for its output without disclosing anything about the secret seed, in a way similar to non-interactive zero-knowledge proofs of correctness.
Efficient implementations of VRFs have not been easy to find, but the literature nowadays contains a number of reasonable constructions under broadly accepted hardness assumptions [226, 183]. In

practice, when adopting the random-oracle model, VRFs can immediately be obtained from unique signatures such as ordinary RSA signatures [226].

Protocol Mastercrypt: Replication with cryptographic randomness from a VRF. With master-slave replication, cryptographically strong randomness secure against faulty non-leader processes can be obtained from a VRF as follows. Initially every process generates a VRF-seed and a verification key. Then it passes the verification key to a trusted entity, which distributes the n verification keys to all processes consistently, ensuring that all correct processes use the same list of verification keys. At every place where the application needs to generate (pseudo-)randomness, the VRF is used by the master to produce the random bits and all processes verify that the bits are unique. Details of this protocol can be found in the full version [85].

3.6 Conclusion

This chapter has introduced a distinction between three models for dealing with non-deterministic operations in BFT replication: modular where the application is a black box; master-slave that needs internal access to the application; and cryptographically secure handling of non-deterministic randomness generation. In the past, dedicated BFT replication systems have often argued for using the master-slave model, but we have learned in the context of distributed applications that changes of the code and understanding an application's logic can be difficult. Hence, our novel Protocol Sieve provides a modular solution that does not require any manual intervention. For a BFT-based state-machine replication platform, Sieve can simply be run without incurring large overhead as a defense against non-determinism, which may be hidden in smart contracts.

Chapter 4 XFT: Practical Fault Tolerance Beyond Crashes

Despite years of intensive research, Byzantine fault-tolerant (BFT) systems have not yet been adopted in practice. This is due to additional cost of BFT in terms of resources, protocol complexity and performance, compared with crash fault-tolerance (CFT). This overhead of BFT comes from the assumption of a powerful adversary that can fully control not only the Byzantine faulty machines, but at the same time also the message delivery schedule across the entire network, effectively inducing communication asynchrony and partitioning otherwise correct machines at will. To many practitioners, however, such strong attacks appear irrelevant.
In this chapter, we introduce cross fault tolerance or XFT, a novel approach to building reliable and secure distributed systems and apply it to the classical state-machine replication (SMR) problem. In short, an XFT SMR protocol provides the reliability guarantees of widely used asynchronous CFT SMR protocols such as Paxos and Raft, but also tolerates Byzantine faults in combination with network asynchrony, as long as a majority of replicas are correct and communicate synchronously. This allows the development of XFT systems at the price of CFT (already paid for in practice), yet with strictly stronger resilience than CFT — sometimes even stronger than BFT itself.
As a showcase for XFT, we present XPaxos, the first XFT SMR protocol, and deploy it in a geo-replicated setting. Although it offers much stronger resilience than CFT SMR at no extra resource cost, the performance of XPaxos matches that of the state-of-the-art CFT protocols.

4.1 Background

Tolerance to any kind of service disruption, whether caused by a simple hardware fault or by a large-scale disaster, is key for the survival of modern distributed systems. Cloud-scale applications must be inherently resilient, as any outage has direct implications on the business behind them [203].
Modern production systems (e.g., [117, 87]) increase the number of nines of reliability[1] by employing sophisticated distributed protocols that tolerate crash machine faults as well as network faults, such as network partitions or asynchrony, which reflect the inability of otherwise correct machines to communicate among each other in a timely manner. At the heart of these systems typically lies a crash fault-tolerant (CFT) consensus-based state-machine replication (SMR) primitive [278].
These systems cannot deal with non-crash (or Byzantine [212]) faults, which include not only malicious, adversarial behavior, but also arise from errors in the hardware, stale or corrupted data from storage systems, memory errors caused by physical effects, bugs in software, hardware faults due to ever smaller circuits, and human mistakes that cause state corruptions and data loss. However, such problems do occur in practice — each of these faults has a public record of taking down major production systems and corrupting their service [118, 47].
Despite more than 30 years of intensive research since the seminal work of Lamport, Shostak and Pease [212], no *practical* answer to tolerating non-crash faults has emerged so far. In particular, asynchronous Byzantine fault-tolerance (BFT), which promises to resolve this problem [96], has not lived up to this expectation, largely because of its extra cost compared with CFT. Namely, asynchronous (that is, "eventually synchronous" [141]) BFT SMR must use at least $3t + 1$ replicas to tolerate t

[1] As an illustration, five nines reliability means that a system is up and correctly running at least 99.999% of the time. In other words, malfunction is limited to one hour every 10 years on average.

non-crash faults [74] instead of only $2t + 1$ replicas for CFT, as used by Paxos [210] or Raft [251], for example.
The overhead of asynchronous BFT is due to the extraordinary power given to the adversary, which may control both the Byzantine faulty machines and the entire network in a coordinated way. In particular, the classical BFT adversary can partition any number of otherwise correct machines at will. In line with observations by practitioners [205], we claim that this adversary model is actually too strong for the phenomena observed in deployed systems. For instance, accidental non-crash faults usually do not lead to network partitions. Even malicious non-crash faults rarely cause the whole network to break down in wide-area networks and geo-replicated systems. The proverbial all-powerful attacker as a common source behind those faults is a popular and powerful simplification used for the design phase, but it has not seen equivalent proliferation in practice.
In this chapter, we introduce XFT (short for cross fault tolerance), a novel approach to building efficient resilient distributed systems that tolerate both non-crash (Byzantine) faults and network faults (asynchrony). In short, XFT allows building resilient systems that

- do not use extra resources (replicas) compared with asynchronous CFT;
- preserve all reliability guarantees of asynchronous CFT (that is, in the absence of Byzantine faults); and
- provide correct service (i.e., safety and liveness [34]) even when Byzantine faults do occur, as long as a majority of the replicas are correct and can communicate with each other synchronously (that is, when a minority of the replicas are Byzantine-faulty or partitioned because of a network fault).

In particular, we envision XFT for wide-area or geo-replicated systems [117], as well as for any other deployment where an adversary cannot easily coordinate enough network partitions and Byzantine-faulty machine actions at the same time.
As a showcase for XFT, we present XPaxos, the first state-machine replication protocol in the XFT model. XPaxos tolerates faults beyond crashes in an efficient and practical way, achieving much greater coverage of realistic failure scenarios than the state-of-the-art CFT SMR protocols, such as Paxos or Raft. This comes without resource overhead as XPaxos uses $2t+1$ replicas. To validate the performance of XPaxos, we deployed it in a geo-replicated setting across Amazon EC2 datacenters worldwide. In particular, we integrated XPaxos within Apache ZooKeeper, a prominent and widely used coordination service for cloud systems [181]. Our evaluation on EC2 shows that XPaxos performs almost as well in terms of throughput and latency as a WAN-optimized variant of Paxos, and significantly better than the best available BFT protocols. In our evaluation, XPaxos even outperforms the native CFT SMR protocol built into ZooKeeper [189].
Finally, and perhaps surprisingly, we show that XFT can offer strictly stronger reliability guarantees than state-of-the-art BFT, for instance under the assumption that machine faults and network faults occur as independent and identically distributed random variables, for certain probabilities. To this end, we calculate the number of nines of consistency (system safety) and availability (system liveness) of resource-optimal CFT, BFT and XFT (e.g., XPaxos) protocols. Whereas XFT always provides strictly stronger consistency and availability guarantees than CFT and always strictly stronger availability guarantees than BFT, our reliability analysis shows that, in some cases, XFT also provides strictly stronger consistency guarantees than BFT.
The remainder of this chapter is organized as follows. In Section 4.2, we define the system model, which is then followed by the definition of the XFT model in Section 4.3. In Section 4.4 and Section 4.5, we present XPaxos and its evaluation in the geo-replicated context, respectively. Section 4.6 provides simplified reliability analysis comparing XFT with CFT and BFT. We overview related work and conclude in Section 4.7.

4.2 System model

Machines. We consider a message-passing distributed system containing a set Π of $n = |\Pi|$ machines , also called replicas . Additionally, there is a separate set C of client machines.
Clients and replicas may suffer from Byzantine faults: we distinguish between crash faults, where a machine simply stops all computation and communication, and non-crash faults, where a machine acts arbitrarily, but cannot break cryptographic primitives we use (cryptographic hashes, MACs, message digests and digital signatures). A machine that is not faulty is called correct. We say a machine is benign if the machine is correct or crash-faulty. We further denote the number of replica faults at a given moment s by

- $t_c(s)$: the number of crash-faulty replicas, and
- $t_{nc}(s)$: the number of non-crash-faulty replicas.

Network. Each pair of replicas is connected with reliable point-to-point bi-directional communication channels. In addition, each client can communicate with any replica.
The system can be asynchronous in the sense that machines may not be able to exchange messages and obtain responses to their requests in time. In other words, network faults are possible; we define a network fault as the inability of some correct replicas to communicate with each other in a timely manner, that is, when a message exchanged between two correct replicas cannot be delivered and processed within delay Δ, known to all replicas. Note that Δ is a deployment specific parameter: we discuss practical choices for Δ in the context of our geo-replicated setting in Section 6.4. Finally, we assume an eventually synchronous system in which, eventually, network faults do not occur [141].
Note that we model an excessive processing delay as a network problem and not as an issue related to a machine fault. This choice is made consciously, rooted in the experience that for the general class of protocols considered in this work, a long local processing time is never an issue on correct machines compared with network delays.
To help quantify the number of network faults, we first give the definition of partitioned replica.

Definition 1 (Partitioned replica)**.** *Replica p is partitioned if p is **not** in the largest subset of replicas, in which every pair of replicas can communicate among each other within delay Δ.*

If there is more than one subset with the maximum size, only one of them is recognized as the largest subset. For example in Figure 4.1, the number of partitioned replicas is 3, counting either the group of p_1, p_4 and p_5 or that of p_2, p_3 and p_5. The number of partitioned replicas can be as much as $n - 1$, which means that no two replicas can communicate with each other within delay Δ. We say replica p is synchronous if p is not partitioned. We now quantify network faults at a given moment s as

- $t_p(s)$: the number of correct, but partitioned replicas.

Problem. We focus on the deterministic state-machine replication problem (SMR) [278]. In short, in SMR clients invoke requests, which are then committed by replicas. SMR ensures

- safety, or consistency, by (a) enforcing total order across committed client's requests across all correct replicas; and by (b) enforcing validity, i.e., that a correct replica commits a request only if it was previously invoked by a client;
- liveness, or availability, by eventually committing a request by a correct client at all correct replicas and returning an application-level reply to the client.

4.3 The XFT model

This section introduces the XFT model and relates it to the established crash-fault tolerance (CFT) and Byzantine-fault tolerance (BFT) models.

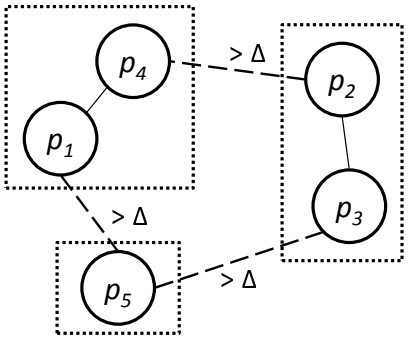


Figure 4.1: An illustration of partitioned replicas: $\{p_1, p_4, p_5\}$ or $\{p_2, p_3, p_5\}$ are partitioned based on Definition 1.

4.3.1 XFT in a nutshell

		Maximum number of each type of replica faults		
		non-crash faults	crash faults	partitioned replicas
Asynchronous CFT (e.g., Paxos [211])	consistency	0	n	$n-1$
	availability	0	$\lfloor \frac{n-1}{2} \rfloor$ (combined)	
Asynchronous BFT (e.g., PBFT [96])	consistency	$\lfloor \frac{n-1}{3} \rfloor$	n	$n-1$
	availability	$\lfloor \frac{n-1}{3} \rfloor$ (combined)		
(Authenticated) Synchronous BFT (e.g., [212])	consistency	$n-1$	n	0
	availability	$n-1$ (combined)		0
XFT (e.g., XPaxos)	consistency	0	n	$n-1$
		$\lfloor \frac{n-1}{2} \rfloor$ (combined)		
	availability	$\lfloor \frac{n-1}{2} \rfloor$ (combined)		

Table 4.1: The maximum numbers of each type of fault tolerated by representative SMR protocols. Note that XFT provides consistency in two modes, depending on the occurrence of non-crash faults.

Classical CFT and BFT explicitly model machine faults only. These are then combined with an orthogonal network fault model, either the synchronous model (where network faults in our sense are ruled out), or the asynchronous model (which includes any number of network faults). Hence, previous work can be classified into four categories: synchronous CFT [120, 278], asynchronous CFT [278, 210, 249], synchronous BFT [212, 135, 64], and asynchronous BFT [96, 44].
XFT, in contrast, redefines the boundaries between machine and network fault dimensions: XFT allows the design of reliable protocols that tolerate crash machine faults regardless of the number of network faults and that, at the same time, tolerate non-crash machine faults when the number of machines that are either faulty or partitioned is within a threshold.
To formalize XFT, we first define anarchy, a very severe system condition with actual non-crash machine (replica) faults and plenty of faults of different kinds, as follows:

Definition 2 (Anarchy)**.** *The system is in anarchy at a given moment s iff $t_{nc}(s) > 0$ and $t_c(s) + t_{nc}(s) + t_p(s) > t$.*

Here, t is the threshold of replica faults, such that $t \leq \lfloor \frac{n-1}{2} \rfloor$. In other words, in anarchy, some replica is non-crash-faulty, and there is no correct and synchronous majority of replicas. Armed with the definition of anarchy, we can define XFT protocols for an arbitrary distributed computing problem in function of its safety property [34].

Definition 3 (XFT protocol)**.** *Protocol P is an XFT protocol if P satisfies safety in all executions in which the system is never in anarchy.*

Liveness of an XFT protocol will typically depend on a problem and implementation. For instance, for deterministic SMR we consider, our XPaxos protocol eventually satisfies liveness, provided a majority of replicas is correct and synchronous. This can be shown optimal.

4.3.2 XFT vs. CFT/BFT

Table 4.1 illustrates differences between XFT and CFT/BFT in terms of their consistency and availability guarantees for SMR.
State-of-the-art asynchronous CFT protocols [211, 251] guarantee consistency despite any number of crash-faulty replicas and any number of partitioned replicas. They also guarantee availability whenever a majority of replicas ($t \leq \lfloor \frac{n-1}{2} \rfloor$) are correct and synchronous. As soon as a single machine is non-crash-faulty, CFT protocols guarantee neither consistency nor availability.
Optimal asynchronous BFT protocols [96, 197, 44] guarantee consistency despite any number of crash-faulty or partitioned replicas, with at most $t = \lfloor \frac{n-1}{3} \rfloor$ non-crash-faulty replicas. They also guarantee availability with up to $\lfloor \frac{n-1}{3} \rfloor$ combined faults, i.e., whenever more than two-thirds of replicas are correct and not partitioned. Note that BFT availability might be weaker than that of CFT in the absence of non-crash faults — unlike CFT, BFT does not guarantee availability when the sum of crash-faulty and partitioned replicas is in the range $[n/3, n/2)$.
Synchronous BFT protocols (e.g., [212]) do not consider the existence of correct, but partitioned replicas. This makes for a very strong assumption — and helps synchronous BFT protocols that use digital signatures for message authentication (so called authenticated protocols) to tolerate up to $n-1$ non-crash-faulty replicas.
In contrast, XFT protocols with optimal resilience, such as our XPaxos, guarantee consistency in two modes: (i) without non-crash faults, despite any number of crash-faulty and partitioned replicas (i.e., just like CFT), and (ii) with non-crash faults, whenever a majority of replicas are correct and not partitioned, i.e., provided the sum of all kinds of faults (machine or network faults) does not exceed $\lfloor \frac{n-1}{2} \rfloor$. Similarly, it also guarantees availability whenever a majority of replicas are correct and not partitioned.
It may be tempting to view XFT as some sort of a combination of the asynchronous CFT and synchronous BFT models. However, this is misleading, as even with actual non-crash faults, XFT is incomparable to authenticated synchronous BFT. Specifically, authenticated synchronous BFT protocols, such as the seminal Byzantine Generals protocol [212], may violate consistency with a single partitioned replica. For instance, with $n = 5$ replicas and an execution in which three replicas are correct and synchronous, one replica is correct but partitioned and one replica is non-crash-faulty, the XFT model mandates that the consistency be preserved, whereas the Byzantine Generals protocol may violate consistency.[2]
Furthermore, from Table 4.1, it is evident that XFT offers strictly stronger guarantees than asynchronous CFT, for both availability and consistency. XFT also offers strictly stronger availability guarantees than asynchronous BFT. Finally, the consistency guarantees of XFT are incomparable to those of asynchronous BFT. On the one hand, outside anarchy, XFT is consistent with the number of non-crash faults in the range $[n/3, n/2)$, whereas asynchronous BFT is not. On the other hand, unlike XFT, asynchronous BFT is consistent in anarchy provided the number of non-crash faults is less than $n/3$. We discuss these points further in Section 4.6, where we also quantify the reliability comparison between XFT and asynchronous CFT/BFT assuming the special case of independent faults.

[2]XFT is not stronger than authenticated synchronous BFT either, as the latter tolerates more machine faults in the complete absence of network faults.

4.3.3 Where to use XFT?

The intuition behind XFT starts from the assumption that "extremely bad" system conditions, such as anarchy, are very rare, and that providing consistency guarantees in anarchy might not be worth paying the asynchronous BFT premium.
In practice, this assumption is plausible in many deployments. We envision XFT for use cases in which an adversary cannot easily coordinate enough network partitions and non-crash-faulty machine actions at the same time. Some interesting candidate use cases include:

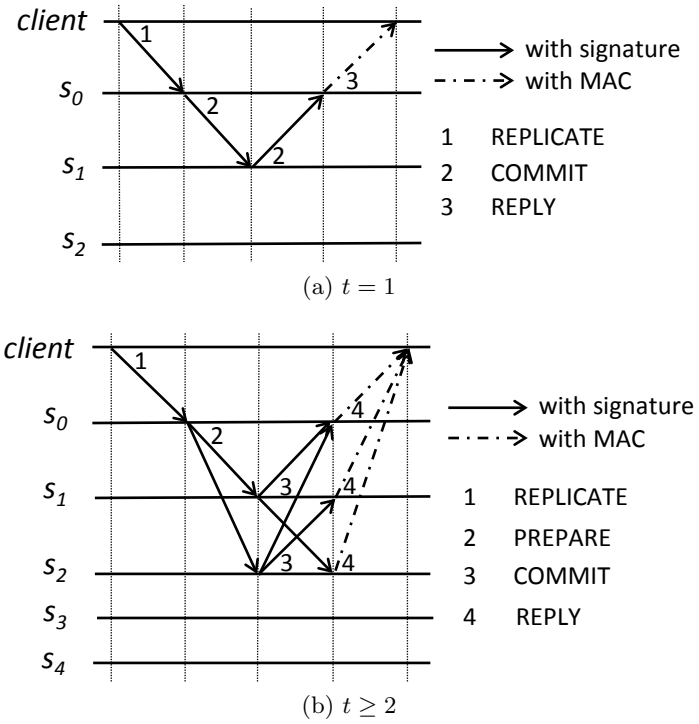
- Tolerating "accidental" non-crash faults. In systems which are not susceptible to malicious behavior and deliberate attacks, XFT can be used to protect against "accidental" non-crash faults, which can be assumed to be largely independent of network faults. In such cases, XFT could be used to harden CFT systems without considerable overhead of BFT.
- Wide-area networks and geo-replicated systems. XFT may reveal useful even in cases where the system is susceptible to malicious non-crash faults, as long as it may be difficult or expensive for an adversary to coordinate an attack to compromise Byzantine machines and partition sufficiently many replicas at the same time. Particularly interesting for XFT are WAN and geo-replicated systems which often enjoy redundant communication paths and typically have a smaller surface for network-level DoS attacks (e.g., no multicast storms and flooding).
- Blockchain. A special case of geo-replicated systems, interesting to XFT, are blockchain systems. In a typical blockchain system, such as Bitcoin [243], participants may be financially motivated to act maliciously, yet may lack the means and capabilities to compromise the communication among (a large number of) correct participants. In this context, XFT is particularly interesting for so-called permissioned blockchains, which are based on state-machine replication rather than on Bitcoin-style proof-of-work [309].

4.4 XPaxos Protocol

In a nutshell, XPaxos consists of two main components:

- a common-case protocol, which replicates and totally orders requests across replicas; this has, roughly speaking, the message pattern and complexity of communication among replicas of state-of-the-art CFT protocols (e.g., Phase 2 of Paxos), hardened by the use of digital signatures;
- a novel view change protocol, in which the information is transferred from one view (system configuration) to another in a decentralized, leaderless fashion.

XPaxos is orchestrated in a sequence of views [96]. The central idea in XPaxos is that, during common-case operation in a given view, XPaxos synchronously replicates clients' requests to only $t+1$ replicas, which are the members of a synchronous group (out of $n = 2t+1$ replicas in total). Each view number i uniquely determines the synchronous group, sg_i, using a mapping known to all replicas. Every synchronous group consists of one primary and t followers, which are jointly called active replicas. Remaining t replicas in a given view are called passive replicas; optionally, passive replicas learn the order from the active replicas using the lazy replication approach [206]. A view is not changed unless there is a machine or network fault within the synchronous group.
In the common case (Section 4.4.1), the clients send digitally signed requests to the primary which are then replicated across $t+1$ active replicas. These $t+1$ replicas digitally sign and locally log the proofs for all replicated requests to their commit logs. Commit logs then serve as the basis for maintaining consistency in view changes.
XPaxos view change (Section 4.4.2), reconfigures the entire synchronous group and not only the leader. All $t+1$ active replicas from the new synchronous group sg_{i+1} try to transfer the state from preceding views to view $i+1$. This decentralized approach to view change is in sharp contrast to classical



(a) $t = 1$

(b) $t \geq 2$

Figure 4.2: XPaxos common-case message patterns for $t = 1$ and $t \geq 2$ (here $t = 2$). Synchronous group illustrated are (s_0,s_1) (when $t = 1$) and (s_0,s_1,s_2) (when $t = 2$), respectively.

reconfiguration/view-change in CFT and BFT protocols (e.g., [210, 96]), in which only a single replica (the primary) leads the view change and transfers the state from previous views. This difference is crucial to maintaining consistency (i.e., linearizability) across XPaxos views in the presence of non-crash faults (but in the absence of full anarchy), despite replicating only across $t + 1$ replicas in the common case. XPaxos novel and decentralized view-change scheme guarantees that, even in presence of non-crash faults, but outside anarchy, at least one correct replica from the new synchronous group sg_{i+1} will be able to transfer the correct state from previous views, as it will be able to contact some correct replicas from old synchronous groups.

Besides, we specially design a fault detection (FD) mechanism, which can help detect, outside anarchy, non-crash faults that would leave the system in an inconsistent state in anarchy. The FD mechanism serves to minimize the impact of long-lived non-crash faults in the system and help detect them before they coincide with a sufficient number of crash faults and network faults to push the system into anarchy.

The main idea behind the FD scheme of XPaxos is the following. In view change, a non-crash faulty replica (of an old synchronous group) might omit to transfer its latest state to a correct replica in the new synchronous group. This fault is dangerous, as it may violate consistency when the system is in anarchy. However, such a fault can be detected using digital signatures from the commit log of some correct replicas (from an old synchronous group), provided that such correct replicas can synchronously communicate with correct replicas from the new synchronous group. In a sense, with XPaxos FD, a critical non-crash machine fault must occur for the first time together with enough crash or partitioned machines (i.e., in anarchy) to violate consistency.

In the following, we explain the core of XPaxos for the common-case (Sec. 4.4.1) and view-change (Sec. 4.4.2) components. XPaxos correctness arguments are given in Sec. 4.4.3.

4.4.1 Common case

Figure 4.2 gives XPaxos common-case message patterns in the special case when $t = 1$ and in the general case when $t \geq 2$. XPaxos is specifically optimized for the case where $t = 1$, as in this case,

there are only two active replicas in each view. The special case $t = 1$ is also very relevant in practice (see e.g., Google Spanner [117]). In the following, we denote the digest of a message m by $D(m)$, whereas $\langle m \rangle_{\sigma_p}$ denotes a message that contains both $D(m)$ signed by the private key of machine p and m.

Tolerating a single fault. For $t = 1$ (see Fig. 4.2a), the XPaxos common case involves only 2 messages between 2 active replicas. Upon receiving a signed request $req = \langle \text{REPLICATE}, op, ts_c, c \rangle_{\sigma_c}$ from client c (where op is the client's operation and ts_c is the clients' timestamp), the primary (say s_0) increments the sequence number sn, signs sn along the digest of req and view number i in message $m_0 = \langle \text{COMMIT}, D(req), sn, i \rangle_{\sigma_{s_0}}$, stores $\langle req, m_0 \rangle$ into its prepare log ($PrepareLog_{s_0}[sn] = \langle req, m_0 \rangle$) (we say s_0 prepares req), and sends the message $\langle req, m_0 \rangle$ to the follower, say s_1.
On receiving $\langle req, m_0 \rangle$, the follower s_1 verifies the client's and primary's signatures, and checks if its local sequence number equals $sn - 1$. Then, the follower updates its local sequence number to sn, executes the request producing reply $R(req)$, and signs message m_1; m_1 is similar to m_0 yet also includes the client's timestamp and the digest of the reply: $m_1 = \langle \text{COMMIT}, \langle D(req), sn, i, req.ts_c, D(R(req)) \rangle_{\sigma_{s_1}}$. The follower then saves the tuple $\langle req, m_0, m_1 \rangle$ to its commit log ($CommitLog_{s_1}[sn] = \langle req, m_0, m_1 \rangle$) and sends m_1 to the primary. The primary, on receiving a valid COMMIT message from the follower (with a matching entry in its prepare log) executes the request, compares the reply $R(req)$ to the follower's digest contained in m_1, and stores $\langle req, m_0, m_1 \rangle$ in its commit log. Finally, it returns an authenticated reply containing m_1 to c, which commits the request if all digests and the follower's signature match.

General case. In case $t \geq 2$, the common-case message pattern of XPaxos (see Fig. 4.2b) contains an explicit PREPARE phase. More specifically, the primary (s_0) assigns a sequence number sn to a client's signed REPLICATE request req and forwards req to all other active replicas (i.e, the t followers) together with the $prep = \langle \text{PREPARE}, D(req), sn, i \rangle_{\sigma_{s_0}}$ message. Each follower verifies the primary's and client's signatures, checks if its local sequence number equals $sn - 1$, and logs $\langle req, prep \rangle$ into its prepare log $PrepareLog_*[sn]$. Then, a follower updates its local sequence number, signs the digest of the request, the view number and the sequence number, and forwards it to all active replicas within a COMMIT message. Upon receiving t signed COMMIT messages — one from each follower — (with a matching entry in the prepare log), an active replica logs $prep$ and the t signed COMMIT messages into its commit log $CommitLog_*[sn]$. Finally, each active replica executes the request and sends the reply to the client (followers may only send the digest of the reply). The client commits the request when it receives matching REPLY messages from all $t + 1$ active replicas.
In both cases ($t = 1$ and $t \geq 2$), a client that timeouts without committing the requests, broadcasts the request to all replicas. Active replicas then forward such request to the primary and trigger a retransmission timer within which a correct active replica expects the client's request to be committed.

4.4.2 View change

Intuition. The ordered requests in commit logs of correct replicas are the key to enforcing consistency (total order) in XPaxos. To illustrate XPaxos view change, consider synchronous groups sg_i and sg_{i+1} of views i and $i + 1$, respectively, each containing $t + 1$ replicas. Proofs of requests committed in sg_i might be logged by as few as a single correct replica in sg_i. Nevertheless, XPaxos view change must ensure that (outside anarchy) these proofs are transferred to the new view $i + 1$. To this end, we had to depart from traditional view change techniques [96, 197, 113] where the entire view change is led by a single replica, usually the primary of the new view. Namely, in XPaxos view-change, every active replica in sg_{i+1} retrieves information about requests committed in preceding views. Intuitively, at least one correct replica from sg_{i+1} will contact (at least one) correct replica from sg_i and transfer the latest correct commit log to the new view $i + 1$.
In the following, we first describe how we choose active replicas for each view. Then, we explain how view changes are initiated, and, finally, how view changes are performed.

		Synchronous Groups ($i \in \mathbb{N}_0$)		
		sg_i	sg_{i+1}	sg_{i+2}
Active replicas	Primary	s_0	s_0	s_1
	Follower	s_1	s_2	s_2
Passive replica		s_2	s_1	s_0

Table 4.2: Synchronous group combinations ($t = 1$).

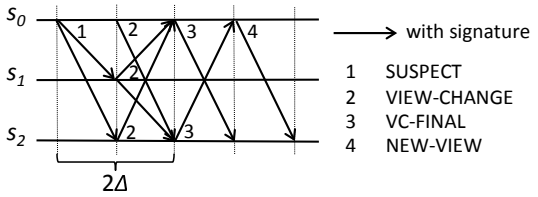


Figure 4.3: XPaxos view change illustration: synchronous group is changed from (s_0,s_1) to (s_0,s_2).

4.4.2.1 Choosing active replicas

To choose active replicas for view i, we enumerate all sets containing $t + 1$ replicas (i.e., $\binom{2t+1}{t+1}$ sets) which then alternate as synchronous groups across views in a round robin fashion. Additionally, each synchronous group uniquely determines the primary. We assume that the mapping from view numbers to synchronous groups is known to all replicas (see e.g., Table 4.2).

4.4.2.2 View change initiation

If a synchronous group in view i (denoted by sg_i) does not make progress, XPaxos performs a view change. Only an active replica of sg_i may initiate a view change.
An active replica $s_j \in sg_i$ initiates a view change if: (i) s_j receives a message from another active replica that does not conform to the protocol (e.g., an invalid signature), (ii) the retransmission timer at s_j expires, (iii) s_j does not complete a view change to view i in a timely manner, or (iv) s_j receives a valid SUSPECT message for view i from another replica. Upon view change initiation, s_j stops participating in the current view and sends $\langle \text{SUSPECT}, i, s_j \rangle_{\sigma_{s_j}}$ to all other replicas.

4.4.2.3 Performing view-change

Upon receiving SUSPECT message from active replica in view i (see the message pattern in Fig. 4.3), replica s_j stops processing messages of view i and sends $m = \langle \text{VIEW-CHANGE}, i+1, s_j, CommitLog_{s_j} \rangle_{\sigma_{s_j}}$ to $t + 1$ replicas in sg_{i+1}. A VIEW-CHANGE message contains the commit log $CommitLog_{s_j}$ of s_j. Commit logs might be empty (e.g., if s_j was passive). Moreover, s_j sends the SUSPECT message to all replicas to accelerate the view change.
Note that XPaxos requires all active replicas in new view to collect the most recent state and its proof (i.e., VIEW-CHANGE messages), rather than the new primary only. Otherwise, a faulty new primary could purposely omit the VIEW-CHANGE message which contains the most recent state, even outside anarchy. Active replica s_j in view $i + 1$ waits for at least $n - t$ VIEW-CHANGE messages from all, but also waits for 2Δ time trying to collect messages as many as possible.
Upon completion of the above protocol, each active replica $s_j \in sg_{i+1}$ inserts all VIEW-CHANGE messages it has received in set $VCSet_{s_j}^{i+1}$. Then s_j sends $\langle \text{VC-FINAL}, i + 1, s_j, VCSet_{s_j}^{i+1} \rangle_{\sigma_{s_j}}$ to every active replica in view $i+1$. This serves to exchange the received VIEW-CHANGE messages among active replicas.
Every active replica $s_j \in sg_{i+1}$ must receive VC-FINAL messages from all active replicas in sg_{i+1}, after which s_j enriches $VCSet_{s_j}^{i+1}$ by combining $VCSet_{*}^{i+1}$ sets piggybacked in VC-FINAL messages. Then,

for each sequence number sn, active replicas select the commit log with the highest view number in all VIEW-CHANGE messages, to confirm the committed request at sn (might be null).
Afterwards, to prepare and commit selected requests in view $i+1$, the new primary ps_{i+1} sends $\langle \text{NEW-VIEW}, i+1, PrepareLog \rangle_{\sigma_{ps_{i+1}}}$ to every active replica in sg_{i+1}, where array $PrepareLog$ contains prepare logs generated in view $i+1$ for each selected request. Upon receiving NEW-VIEW message, every active replica $s_j \in sg_{i+1}$ processes prepare logs in $PrepareLog$ as described in the common case (see Sec. 4.4.1).
Finally, every active replica $s_j \in sg_{i+1}$ makes sure that all selected requests in $PrepareLog$ are committed in view $i+1$. When this condition is satisfied, XPaxos can start processing new requests.

4.4.3 Correctness arguments

Consistency (Total Order). XPaxos enforces the following invariant, which is key to total order.

Lemma 1. *Outside anarchy, if a benign client c commits a request req with sequence number sn in view i, and a benign replica s_k commits the request req' with sn in view $i' > i$, then $req = req'$.*

A benign client c commits request req with sequence number sn in view i, only after c receives matching replies from $t+1$ active replicas in sg_i. This implies that every benign replica in sg_i stores req into its commit log under sequence number sn. In the following, we focus on the special case where: $i' = i+1$. This serves as the base step for the proof of Lemma 1 by induction across views that we postpone to [41].
Recall that, in view $i' = i+1$, all (benign) replicas from sg_{i+1} wait for $n - t = t + 1$ VIEW-CHANGE messages containing commit logs transferred from other replicas, as well as the timer set to 2Δ to expire. Then, replicas in sg_{i+1} exchange this information within VC-FINAL messages. Notice that, outside anarchy, there exists at least one correct and synchronous replica in sg_{i+1}, say s_j. Hence, a benign replica s_k that commits req' in view $i+1$ under sequence number sn must have had received VC-FINAL from s_j. In turn, s_j waited for $t+1$ VIEW-CHANGE messages (and timer 2Δ), so it received a VIEW-CHANGE message from some correct and synchronous replica $s_x \in sg_i$ (such a replica exists in sg_i as at most t replicas in sg_i are non-crash faulty or partitioned). As s_x stored req under sn in its commit log in view i, it forwards this information to s_j in a VIEW-CHANGE message and s_j forwards this information to s_k within a VC-FINAL. Hence $req = req'$ follows.

Availability. Availability in XPaxos is guaranteed in case the synchronous group contains only correct and synchronous replicas. With eventual synchrony we can assume that, eventually, there will be no network faults. Additionally, with all combinations of $t+1$ replicas (out of $2t+1$) rotating in the role of active replicas, XPaxos guarantees that, eventually, view change in XPaxos will complete with $t+1$ correct and synchronous active replicas.

4.5 Performance Evaluation

	US West (CA)	Europe (EU)	Tokyo (JP)	Sydney (AU)	Sao Paolo (BR)
US East (VA)	88 /1097 /82190 /166390	92 /1112 /85649 /169749	179 /1226 /81177 /165277	268 /1372 /95074 /179174	146 /1214 /85434 /169534
US West (CA)		174 /1184 /1974 /15467	120 /1133 /1180 /6210	186 /1209 /6354 /51646	207 /1252 /90980 /169080
Europe (EU)			287 /1310 /1397 /4798	342 /1375 /3154 /11052	233 /1257 /1382 /9188
Tokyo (JP)				137 /1149 /1414 /5228	394 /2496 /11399 /94775
Sydney (AU)					392 /1496 /2134 /10983

Table 4.3: Round-trip latency of TCP ping (hping3) across Amazon EC2 datacenters, collected during three months. The latencies are given in milliseconds, in the format: average / 99.99% / 99.999% / maximum.

In this section, we evaluate the performance of XPaxos and compare it to Zyzzyva [197], PBFT [96] and a WAN-optimized version of Paxos [210], using the Amazon EC2 worldwide cloud platform. We chose a geo-replicated, WAN settings as we believe that these are a better fit for protocols that tolerate Byzantine faults, including XFT and BFT. Indeed, in WAN settings: (i) there are no single point of

failure such as a switch interconnecting machines, (ii) there are no correlated failures due to, e.g., a power-outage, a storm, or other natural disasters, and (iii) it is difficult for the adversary to flood the network, correlating network and non-crash faults (the last point is relevant for XFT).
In the rest of this section, we first present the experimental setup (Section 4.5.1), and then evaluate the performance (throughput and latency) in the fault-free scenario (Section 4.5.2), as well as under faults (Section 4.5.3). Finally, we perform a performance comparison using a real application: the Zookeeper coordination service [181] (Section 4.5.4), by comparing native Zookeeper to Zookeper variants that use the four above mentioned replication protocols.

4.5.1 Experimental setup

4.5.1.1 Synchrony and XPaxos

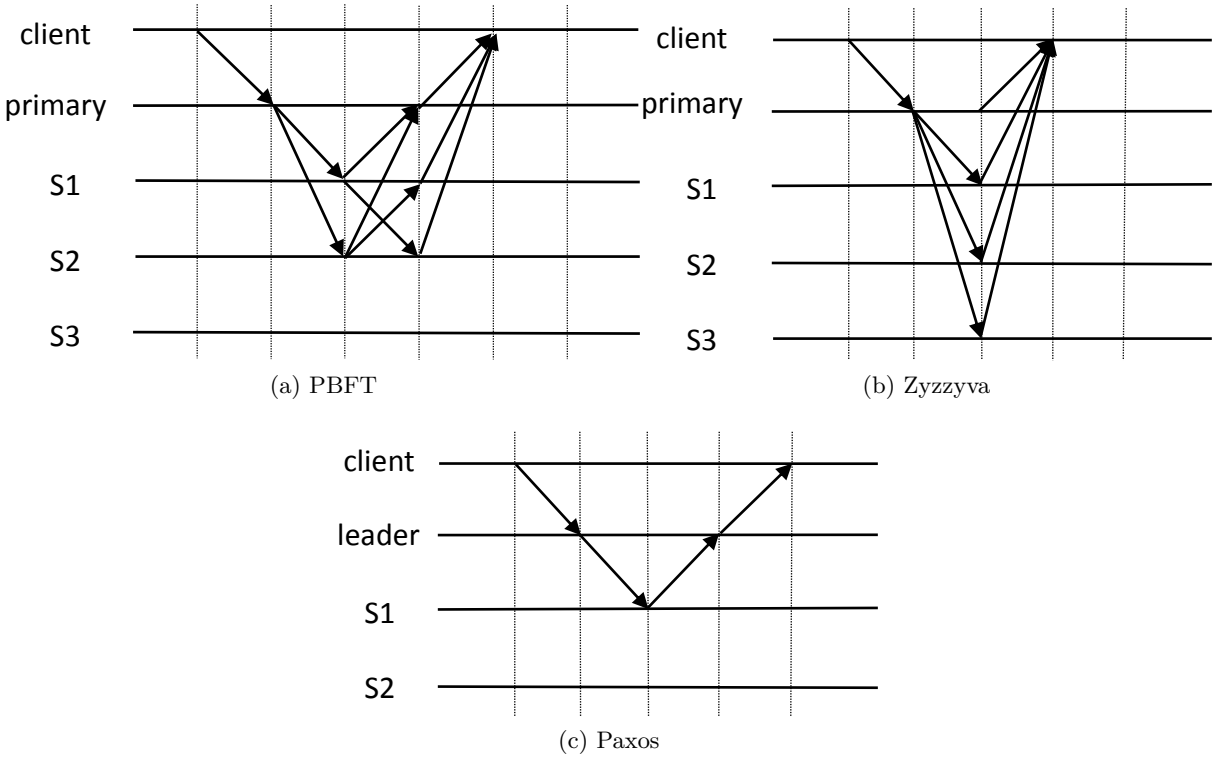
In a practical deployment of XPaxos, a critical parameter is the value of timeout Δ, i.e., the upper bound on communication delay between any two correct machines. If the communication between two correct machines takes more than Δ, we declare a network fault (see Sec. 4.2). Notably, Δ is vital to the XPaxos view-change (Sec. 4.4.2).
To understand the value of Δ in our geo-replicated context, we ran a 3-month experiment during which we continuously measured round-trip latency across six Amazon EC2 datacenters worldwide using TCP ping (hping3). We used the least expensive EC2 micro instances, that arguably have the highest probability of experiencing variable latency due to virtualization. Each instance was pinging all other instances every 100 ms. The results of this experiment are summarized in Table 4.3. While we detected network faults lasting up to 3 minutes, our experiment showed that the round-trip latency between any two datacenters was less than 2.5 seconds 99.99% of the time. Therefore, we adopted the value of $\Delta = 2.5/2 = 1.25$ seconds, yielding $p_{synchrony} = 0.9999$ (i.e., $9_{synchrony} = 4$).

4.5.1.2 Protocols under test

We compare XPaxos against three protocols whose common case message patterns when $t = 1$ are depicted in Figure 4.4. The first two are BFT protocols, namely (a speculative variant of) PBFT [96] and Zyzzyva [197] and require $3t + 1$ replicas to tolerate t faults. We chose PBFT because it is possible to derive a speculative variant of the protocol that relies on a 2-phase common case commit protocol across only $2t + 1$ replicas (Figure 4.4a; see also [96]). In this PBFT variant, the remaining t replicas are not involved in the common case, which is more efficient in a geo-replicated settings. We chose Zyzzyva because it is the fastest BFT protocol that involves all replicas in the common case (Figure 4.4b). The third protocol we compare against is a very efficient WAN-optimized variant of crash-tolerant Paxos inspired by [49, 200, 117]. We have chosen the variant of Paxos that exhibits the fastest write pattern (Figure 4.4c). This variant requires $2t + 1$ replicas to tolerate t faults, but involves $t + 1$ replicas in the common case, i.e., just like XPaxos.
In order to provide a fair comparison, all protocols rely on the same Java code base. We rely on HMAC-SHA1 to compute MACs and RSA1024 to sign and verify signatures.

4.5.1.3 Experimental testbed and benchmarks

We run the experiments on the Amazon EC2 platform that comprises widely distributed datacenters, interconnected by the Internet. Communications between datacenters have a low bandwdith and a high latency. We run the experiments on mid-range virtual machines that contain 8 vCPUs, 15GB of memory, 2 x 80 GB SSD Storage, and run Ubuntu Server 14.04 LTS (PV) with the Linux 3.13.0-24-generic x86_64 kernel.
In the case $t = 1$, Table 4.4 gives the deployment of the different replicas at different datacenters, for each analyzed protocol. Clients are always located in the same datacenter as the (initial) primary



(a) PBFT

(b) Zyzzyva

(c) Paxos

Figure 4.4: Communication patterns of the three protocols under test ($t = 1$).

replica to better emulate what is done in modern geo-replicated systems where clients are served by the closest datacenter [289, 117].[3]
To stress the protocols, we run a microbenchmark that is similar to the one used in [96, 197]. In this microbenchmark, each server replicates a *null* service (this means that there is no execution of requests). Moreover, clients issue requests in closed-loop: a client waits for a reply to its current request before issuing a new request. The benchmark allows varying the request size and the reply size. For space limitations, we only report results for two request sizes (1kB, 4kB) and one reply size (0kB). We refer to these microbenchmarks as 1/0 and 4/0 benchmarks, respectively.

4.5.2 Fault-free performance

We first compare the performance of protocols when $t = 1$ in replica configurations as shown in Table 4.4, using 1/0 and 4/0 microbenchmarks. The results are depicted in Figures 4.5a and 4.5b. On each graph, the X-axis shows throughput (in kops/s), and Y-axis shows latency (in ms).

PBFT	Zyzzyva	Paxos	XPaxos	EC2 Region
Primary	Primary	Primary	Primary	US West (CA)
Active	Active	Active	Follower	US East (VA)
		Passive	Passive	Tokyo (JP)
Passive		-	-	Europe (EU)

Table 4.4: Configurations of replicas. Greyed replicas are not used in the "common" case.

As we can see, in both benchmarks, XPaxos achieves significantly better performance than PBFT and

[3]In practice, modern geo-replicated system, like Spanner [117], use hundreds of CFT SMR instances across different partitions to accommodate for geo-distributed clients.

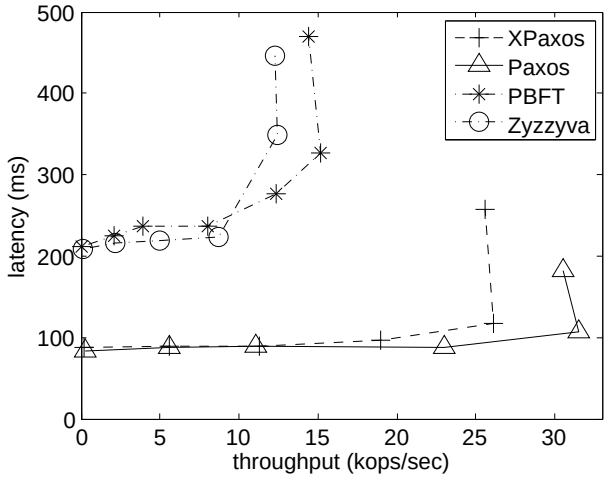
Zyzzyva. Moreover, its performance are very close to that of Paxos. This comes from the fact that in a worldwide cloud environment, network is the bottleneck and message patterns of BFT protocols, namely PBFT and Zyzzyva, reveal costly. Paxos and XPaxos implement a round-trip across two replicas, which renders them very efficient.
Next, to assess the fault scalability of XPaxos, we ran the 1/0 micro-benchmark in configurations that tolerate two faults ($t = 2$). We use the following EC2 datacenters for this experiment: CA (California), OR (Oregon), VA (Virginia), JP (Tokyo), EU (Ireland), AU (Sydney) and SG (Singapore). We place Paxos and XPaxos active replicas at the first $t + 1$ datacenters, and their passive replicas to next t datacenters. PBFT uses the first $2t+1$ datacenters for active replicas and the last t for passive replicas. Finally, Zyzzyva uses all replicas as active replicas.
We observe that XPaxos again clearly outperforms PBFT and Zyzzyva and achieves performance very close to that of Paxos. Moreover, unlike PBFT and Zyzzyva, Paxos and XPaxos only suffer a moderate performance decrease with respect to the $t = 1$ case.

4.5.3 Performance under faults

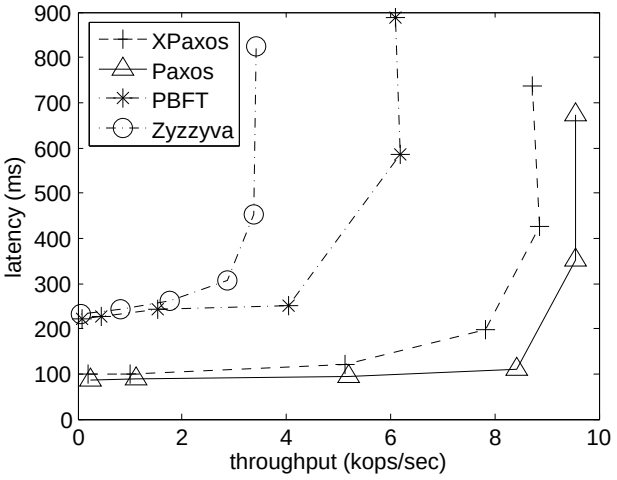
In this section, we analyze the behavior of XPaxos under faults. We run the 1/0 micro-benchmark on three replicas (CA, VA, JP) to tolerate one fault (see also Table 4.4). The experiment starts with CA and VA as active replicas, and with 2500 clients in CA. At time 180s, we crash the follower, VA. At time 300s, we crash the CA replica. At time 420s, we crash the third replica, JP. Each replica recovers 20s after having crashed. Moreover, the timeout 2Δ (used during state transfer in view change, Section 4.4.2) is set to 2.5s (see Sec. 4.5.1.1). We show the throughput of XPaxos in function of time in Figure 4.6, which also indicates active replicas in each view. We observe that after each crash, the system performs a view change that lasts less than 10s, which is very reasonable in a geo-distributed setting. This fast execution of the view change subprotocol is a consequence of XPaxos lazy replication that keeps passive replicas updated. We also observe that XPaxos throughput changes with views. This is because the latency between the primary and the follower, and between the primary and clients, varies from view to view.

4.5.4 Macro-benchmark: ZooKeeper

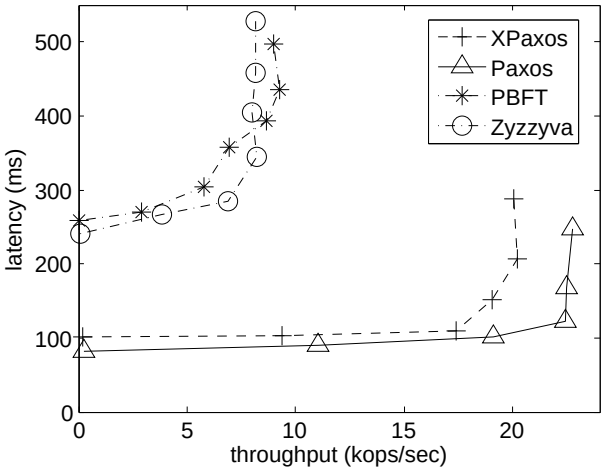
In order to assess the impact of our work on real-life applications, we measured the performance achieved when replicating the ZooKeeper coordination service [181] using all protocols considered in this study: Zyzzyva, PBFT, Paxos and XPaxos. We also compare with the native ZooKeeper performance, when the system is replicated using a built-in replication protocol, called Zab [189]. This protocol is crash-resilient and requires $2t + 1$ replicas to tolerate t faults.
We used the ZooKeeper 3.4.6 codebase. The integration of the various protocols inside ZooKeeper has been carried out by replacing the Zab protocol. For fair comparison to native ZooKeeper, we made a minor modification to native ZooKeeper to force it to use (and keep) a given node as primary. To focus the comparison on performance of replication protocols, and avoid hitting other system bottlenecks (such as storage I/O that is not very efficient in virtualized cloud environments), we store ZooKeeper data and log directories on a volatile tmpfs file system.The tested configuration tolerates one fault ($t = 1$). ZooKeeper clients were located in the same region as the primary replica (CA). Each client invokes 1kB write operations in a closed-loop.
Figure 4.7 depicts the results. The X-axis represents the throughput in kops/sec. The Y-axis represents the latency in ms. As for micro-benchmarks, we observe that Paxos and XPaxos clearly outperform BFT protocols and achieve performance close to that of Paxos. More surprisingly, we can see that XPaxos is more efficient than the built-in Zab protocol, although the latter only tolerates crash faults.



(a) 1/0 benchmark, $t = 1$



(b) 4/0 benchmark, $t = 1$



(c) 1/0 benchmark, $t = 2$

Figure 4.5: Fault-free performance

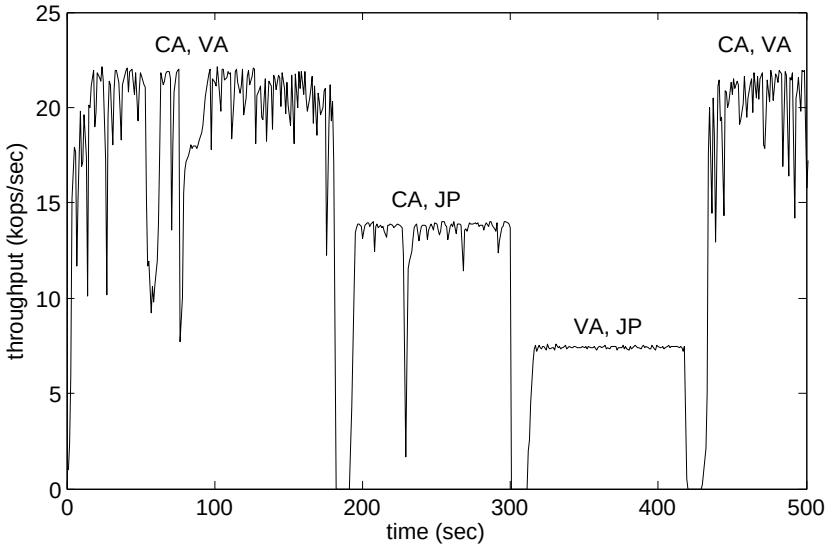


Figure 4.6: XPaxos under faults.

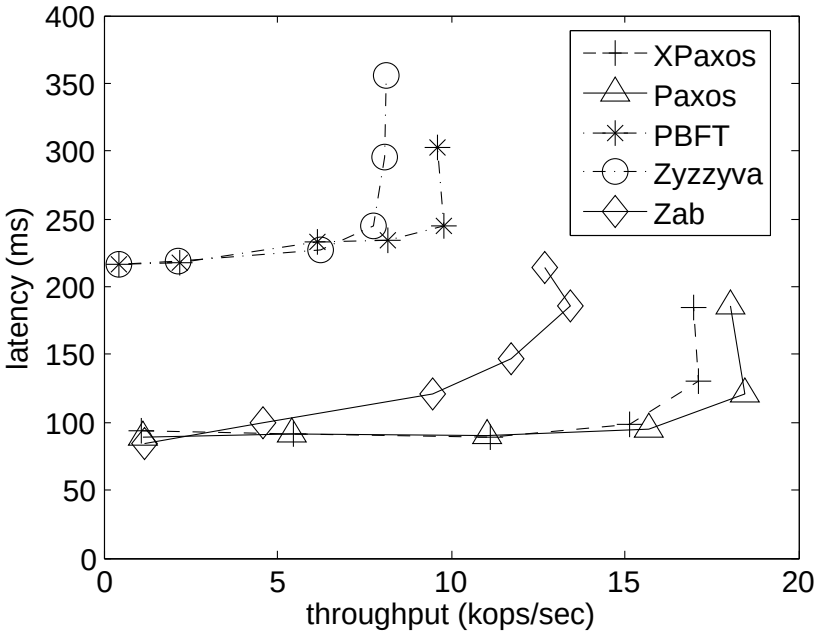


Figure 4.7: Latency vs. throughput for the ZooKeeper application ($t = 1$).

4.6 Reliability Analysis

In order to estimate the coverage of a fault model we consider the fault states of the machines to be independent and identically distributed random variables.
We denote the probability that a replica is correct (resp., crash faulty) by $p_{correct}$ (resp., p_{crash}). The probability that a replica is benign is $p_{benign} = p_{correct} + p_{crash}$. Hence, a replica is non-crash faulty with probability $p_{non\text{-}crash} = 1 - p_{benign}$. Besides, we assume there is a probability $p_{synchrony}$ that a replica is not partitioned, where $p_{synchrony}$ is a function of Δ, the network, and the system environment. Therefore, the probability that a replica is partitioned equals $1 - p_{synchrony}$.
Aligned with the industry practice, we measure reliability guarantees and coverage of fault scenarios using nines of reliability. Specifically, we distinguish nines of consistency and nines of availability and use these measures to compare different fault models. We introduce a function $9of(p)$ that turns a probability p into the corresponding number of "nines", by letting $9of(p) = \lfloor -\log_{10}(1-p) \rfloor$. For example, $9of(0.999) = 3$. For brevity, 9_{benign} stands for $9of(p_{benign})$, and so on, for other probabilities of interest.
We focus here on consistency, as XPaxos guarantees better availability than both CFT and BFT in any case (see Table 4.1).We postpone availability analysis to [41].
We start with the number of nines of consistency for an asynchronous CFT protocol, denoted by $9ofC(CFT) = 9of(P[\text{CFT is consistent}])$. As $P[\text{CFT is consistent}] = p^n_{benign}$, a straightforward calculation yields:

$$9ofC(CFT) = \Big\lfloor -\log_{10}(1 - p_{benign}) - \log_{10}(\sum_{i=0}^{n-1} p^i_{benign}) \Big\rfloor,$$

which gives $9ofC(CFT) \approx 9_{benign} - \lceil \log_{10}(n) \rceil$ for values of p_{benign} close to 1, when p^i_{benign} decreases slowly. As a rule of thumb, for small values of n, i.e., $n < 10$, we have $9ofC(CFT) \approx 9_{benign} - 1$.
In other words, in typical configurations, where few faults are tolerated [117], a CFT system as a whole loses one nine of consistency from the likelihood that a single replica is benign.

4.6.1 XPaxos vs. CFT

We now quantify the advantage of XPaxos over asynchronous CFT. From Table. 4.1, if there is no non-crash fault, or there are no more than t faults (machine faults or network faults), XPaxos is consistent, i.e.,

$$P[\text{XPaxos is consistent}] = p^n_{benign} + \sum_{i=1}^{t=\lfloor \frac{n-1}{2} \rfloor} \binom{n}{i} p^i_{non\text{-}crash}$$
$$\times \sum_{j=0}^{t-i} \binom{n-i}{j} p^j_{crash} \times p^{n-i-j}_{correct} \times \sum_{k=0}^{t-i-j} \binom{n-i-j}{k} \times$$
$$p^{n-i-j-k}_{synchrony} \times (1 - p_{synchrony})^k.$$

To quantify the difference between XPaxos and CFT more tangibly, we calculated $9ofC(\text{XPaxos})$ and $9ofC(CFT)$ for all values of 9_{benign}, $9_{correct}$ and $9_{synchrony}$ ($9_{benign} \geq 9_{correct}$) between 1 and 20 in the special cases where $t = 1$ and $t = 2$, which are most relevant in practice. For $t = 1$, we observed the following relation ($t = 2$ case is postponed to [41]):

$$9ofC(\text{XPaxos}_{t=1}) - 9ofC(CFT_{t=1}) =$$
$$\begin{cases} 9_{correct} - 1, & 9_{benign} > 9_{synchrony} \text{ and} \\ & 9_{synchrony} = 9_{correct}, \\ min(9_{synchrony}, 9_{correct}), & \text{otherwise.} \end{cases}$$

Hence, for $t = 1$ we observe that the number of nines of consistency XPaxos adds on top of CFT is proportional to the nines of probability for correct or synchronous machine. The added nines are not directly related to p_{benign}, although $p_{benign} \geq p_{correct}$ must hold.

Example 1. When $p_{benign} = 0.9999$ and $p_{correct} = p_{synchrony} = 0.999$, we have $p_{non\text{-}crash} = 0.0001$ and $p_{crash} = 0.0009$. In this example, $9 \times p_{non\text{-}crash} = p_{crash}$, i.e., if a machine suffers a faults 10 times, then one of these is a non-crash fault and the rest are crash faults. In this case, $9ofC(CFT_{t=1}) = 9_{benign} - 1 = 3$, whereas $9ofC(\mathsf{XPaxos}_{t=1}) - 9ofC(CFT_{t=1}) = 9_{correct} - 1 = 2$, i.e., $9ofC(\mathsf{XPaxos}_{t=1}) = 5$. XPaxos adds 2 nines of consistency on top of CFT and achieves 5 nines of consistency in total.

Example 2. In a slightly different example, let $p_{benign} = p_{synchrony} = 0.9999$ and $p_{correct} = 0.999$, i.e., the network behaves more reliably than in Example 1. $9ofC(CFT_{t=1}) = 9_{benign} - 1 = 3$, whereas $9ofC(\mathsf{XPaxos}_{t=1}) - 9ofC(CFT_{t=1}) = p_{correct} = 3$, i.e., $9ofC(\mathsf{XPaxos}_{t=1}) = 6$. XPaxos adds 3 nines of consistency on top of CFT and achieves 6 nines of consistency in total.

4.6.2 XPaxos vs. BFT

Recall that (see Table 4.1) SMR in asynchronous BFT model is consistent whenever no more than one-third machines are non-crash faulty. Hence,

$$P[\text{BFT is consistent}] = \sum_{i=0}^{t=\lfloor \frac{n-1}{3} \rfloor} \binom{n}{i} (1 - p_{benign})^i \times p_{benign}^{n-i}.$$

We first examine the conditions under which XPaxos has stronger consistency guarantees than BFT. Fixing the value t of tolerated faults, we observe that $P[\mathsf{XPaxos} \text{ is consistent}] > P[\text{BFT is consistent}]$ is equivalent to:

$$p_{benign}^{2t+1} + \sum_{i=1}^{t} \binom{2t+1}{i} p_{non\text{-}crash}^{i} \times \sum_{j=0}^{t-i} \binom{2t+1-i}{j} p_{crash}^{j} \times$$

$$p_{correct}^{2t+1-i-j} \times \sum_{k=0}^{t-i-j} \binom{2t+1-i-j}{k} p_{synchrony}^{2t+1-i-j-k} \times$$

$$(1 - p_{synchrony})^k > \sum_{i=0}^{t} \binom{3t+1}{i} p_{benign}^{3t+1-i} (1 - p_{benign})^i.$$

In the special case when $t = 1$, the above inequality simplifies to

$$p_{correct} \times p_{synchrony} > p_{benign}^{1.5}.$$

Hence, for $t = 1$, XPaxos has stronger consistency guarantees than any asynchronous BFT protocol whenever the probability that a machine is correct and not partitioned is larger than 1.5 power of the probability that a machine is benign. This is despite the fact that BFT is more expensive than XPaxos as $t = 1$ implies 4 replicas for BFT and only 3 for XPaxos.

In terms of nines of consistency, again for $t = 1$ ($t = 2$ is again postponed to [41]), we calculated the difference in consistency between XPaxos and BFT SMR, for all values of 9_{benign}, $9_{correct}$ and $9_{synchrony}$ ranging between 1 and 20, and observed the following relation:

$$9ofC(BFT_{t=1}) - 9ofC(\mathsf{XPaxos}_{t=1}) =$$

$$\begin{cases} 9_{benign} - 9_{correct} + 1, & 9_{benign} > 9_{synchrony} \ \& \\ & 9_{synchrony} = 9_{correct}, \\ 9_{benign} - min(9_{correct}, 9_{synchrony}), & \text{otherwise.} \end{cases}$$

Notice that in cases where XPaxos guarantees better consistency than BFT ($p_{correct} \times p_{synchrony} > p_{benign}^{1.5}$), it is only "slightly" better and does not materialize in additional nines.
Example 1 (cont'd.). Building upon our example, $p_{benign} = 0.9999$ and $p_{synchrony} = p_{correct} = 0.999$, we have $9ofC(BFT_{t=1}) - 9ofC(\mathsf{XPaxos}_{t=1}) = 9_{benign} - 9_{synchrony} + 1 = 2$, i.e., $9ofC(\mathsf{XPaxos}_{t=1}) = 5$ and $9ofC(BFT_{t=1}) = 7$. BFT brings 2 nines of consistency on top of XPaxos.
Example 2 (cont'd.). When $p_{benign} = p_{synchrony} = 0.9999$ and $p_{correct} = 0.999$, we have $9ofC(BFT_{t=1}) - 9ofC(\mathsf{XPaxos}_{t=1}) = 1$, i.e., $9ofC(\mathsf{XPaxos}_{t=1}) = 6$ and $9ofC(BFT_{t=1}) = 7$. XPaxos has one nine of consistency less than BFT (albeit the only 7th).

4.7 Related work and concluding remarks

In this chapter we introduced XFT, a novel fault-tolerance model that allows design of efficient protocols that tolerate non-crash faults. We demonstrated XFT through XPaxos, a novel state machine replication protocol that features many more nines of reliability than state of the art crash fault-tolerant (CFT) protocols with roughly the same communication complexity, performance and resource cost. Namely, XPaxos uses $2t + 1$ replicas and provides all the reliability guarantees of CFT, yet is also capable of tolerating non-crash faults, so long as a majority of XPaxos replicas are correct and can communicate synchronously among each other.
As XFT is entirely realized in software, it is fundamentally different from an established approach that relies on trusted hardware to reducing the resource cost of BFT to $2t + 1$ replicas only (e.g.,[119, 216, 191, 305]).
XPaxos is also different from PASC [118] approach, which makes CFT protocols tolerate a subset of Byzantine faults using ASC-hardening. ASC-hardening modifies an application by keeping two copies of the state at each replica. It then tolerates Byzantine faults under the "fault diversity" assumption: i.e., a fault will not corrupt both copies of the state in the same way. Unlike XPaxos, PASC does not tolerate Byzantine faults that affect the entire replica (e.g., both state copies).
In this chapter, we did not explore the impact on varying the number of tolerated faults per fault class. In short, this approach, known as the hybrid fault model and introduced in [300] distinguishes the threshold of non-crash faults (say b) despite which safety should be ensured, from the threshold t of faults (of any class) despite which the availability should be ensured (where often $b \leq t$). The hybrid fault model and its refinements [112, 261] appear orthogonal to our XFT approach.
Visigoth Fault Tolerance (VFT) [261] is another recent extension of the hybrid fault model. Besides having different thresholds for non-crash and crash faults, VFT also refines the space between network synchrony and asynchrony by defining the threshold of network faults that a VFT protocol can tolerate. VFT is however different from XFT, in that it fixes separate fault thresholds for non-crash and network faults. This difference is fundamental rather than notational, as XFT cannot be expressed by choosing specific values of VFT thresholds.[4] In addition, VFT protocols have more complex communication patterns than XPaxos. That said, many of the VFT concepts remain orthogonal to XFT. In future, it would be very interesting to explore interactions between the hybrid fault model and its refinements such as VFT with our XFT.
Beyond some research directions outlined above, this work opens more avenues for future work. For instance, many important distributed computing problems beyond SMR, such as distributed storage (see Chapter 7), deserve a novel look through the XFT prism. In addition, we plan to evaluate XFT in the context of blockchain by integrating it with Hyperledger fabric (see Chapter 2).

[4]More specifically, XPaxos can tolerate, with $2t + 1$ replicas, t network faults, t non-crash faults and t crash faults, albeit not simultaneously. Specifying such requirements in VFT would yield at least $3t + 1$ replicas.

Chapter 5 WHEAT: An Empirical Design for Geo-Replicated State Machines

State machine replication (SMR) [208, 278] is a well-known technique to implement fault-tolerant services. The basic idea is to have an arbitrary number of clients issuing requests to a set of replicas in such a way that: (1) all correct replicas execute the same sequence of requests; and (2) clients receive a reply for their requests despite the failure of a fraction of these replicas. This technique is adopted by many modern distributed systems, ranging from cluster-based coordination services (e.g., Chubby [79] and Zookeeper [181]) to geo-replicated databases (e.g., Spanner [117]).

The core of a replicated state machine is an agreement protocol (e.g., Paxos [210]) used to establish a total order in the messages delivered to the replicas. This protocol, which usually requires multiple communication steps, is responsible for a significant latency overhead when SMR is employed for geo-replication, which is the case in many multi-cloud scenarios such as the one we want to consider in Hyperledger and SUPERCLOUD. To mitigate this problem, many SMR protocols have been proposed for wide area networks (WANs) (e.g., [35, 229, 304, 241]). These WAN SMR protocols employ optimizations to reduce latency, usually by decreasing the number of communication steps across the WAN. All these protocols were evaluated in real, emulated or simulated environments, showing the proposed optimizations were indeed effective in decreasing the protocol latency.

However, even though such evaluations generally use comparable methodologies, they do not use the same experimental environments and codebase across independent works. This lack of a common ground makes it hard to not only compare results across distinct papers, but also to assess which optimizations are actually effective in practice. This is aggravated by the fact that these evaluations tend to compare SMR protocols in an holistic manner and generally do not compare individual optimizations.

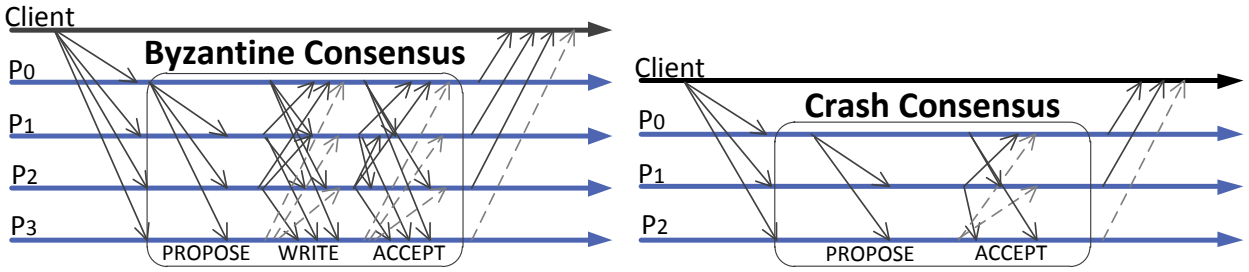
In this chapter we present an extensive evaluation of several latency-related optimizations scattered across the literature (both for local data centers and geo-replication) using the same testbeds, methodology and codebase (see §5.2). More specifically, we selected a subset of optimizations for decreasing the latency of strongly-consistent geo-replicated systems, implemented them in the BFT-SMaRt replication library [68] (see §5.1) and deployed the experiments in the PlanetLab testbed and in the Amazon EC2 cloud. During the course of this evaluation, we obtained some unexpected results. The most notorious example is related with the use of multiple leaders – a widely accepted optimization used by several WAN-optimized protocols such as Mencius [229] and EPaxos [241]. Specifically, our results indicate that this optimization does not bring significant latency reduction just by itself; instead, we observed that using a fixed leader in a fast replica is a more effective (and simpler) strategy to reduce latency. Moreover, we also found that adding a few more replicas to the system without increasing the size of the quorums required by the protocol may lead to significant latency improvements. These results shed light on which optimizations are really effective for improving the latency of geo-replicated state machines, and constitute the first contribution of this work.

The aforementioned results showcasing the benefit of having extra replicas without necessarily increasing the quorum sizes required by the system led to the second contribution of the work: two novel vote (weight) assignment schemes designed to preserve (CFT and BFT) SMR protocol correctness while also allowing the emergence of quorums of variable size (see §5.3.2). By allowing quorums of different sizes, it is possible to avoid the need of accessing a majority of replicas – a requirement of many SMR

protocols. We introduce two vote assignment schemes (for CFT [210] and BFT [96] SMR) and show that they enable the formation of safe and minimal quorums without endangering the consistency and availability of the underlying quorum system [228]. To the best of our knowledge, this is the first work that incorporates the idea of assigning different votes for different replicas (i.e., weighted replication) [162, 262, 157] in replicated state machines.
Our third and final contribution is the design, implementation and evaluation of WHEAT (WeigHt-Enabled Active replicaTion), a flexible WAN-optimized SMR protocol developed by extending BFT-SMaRt with the most effective optimizations (according to our experiments) and our vote assignment schemes (see §5.3.1). The evaluation of WHEAT – conducted in Amazon EC2 (see §5.4) – shows that this protocol could outperform BFT-SMaRt by up to 56% in terms of latency. WHEAT was designed to operate both under crash and Byzantine faults. To the best of our knowledge, WHEAT is the first SMR protocol that is both optimized for geo-replication and capable of withstanding general Byzantine faults; Mencius [229] and EPaxos [241] tolerate only crash faults while BFT protocols like EBAWA [304] or Steward [35] requires either each replica to have a trusted component that can only fail by crash, or only tolerate Byzantine faults within a site (i.e., do not tolerate compromised sites), respectively.

5.1 State Machine Replication & BFT-SMaRt

In the SMR model[208, 278] an arbitrary number of clients send requests to a set of servers, which hosts replicas of a stateful service that updates its state after processing those requests. The goal of this technique is to make the state at each replica evolve in a consistent way, resulting in a service which is accurately replicated across all replicas. Since the service state was already updated by the time clients receive a reply from the service, this technique is able to offer strong consistency, i.e., linearizability [175]. To enforce this behavior, it is necessary that: (1) client requests are delivered to the replicas via total order broadcast; (2) replicas start their execution in the same state; and (3) replicas modify their state in a deterministic way.
All the experimental work done in this work is based on the BFT-SMaRt open-source library [68]. BFT-SMaRt implements a modular SMR protocol on top of a Byzantine consensus algorithm [288]. Under favourable network conditions and the absence of faulty replicas BFT-SMaRt executes the message pattern depicted in Fig.5.1a, which is similar to the PBFT protocol [96].
Clients send their requests to all replicas, triggering the execution of the consensus protocol. Each consensus instance i begins with one replica – the leader – proposing a batch of requests to be decided within that consensus. This is done by sending a PROPOSE message containing the aforementioned batch to the other replicas. All replicas that receive the PROPOSE message verify if its sender is the leader and if the batch proposed is valid. If this is the case, they register the batch being proposed and send a WRITE message to all other replicas containing a cryptographic hash of the proposed batch. If a replica receives $\lceil \frac{n+f+1}{2} \rceil$ WRITE messages with the same hash, it sends an ACCEPT message to all other replicas containing this hash. If some replica receives $\lceil \frac{n+f+1}{2} \rceil$ ACCEPT messages for the same hash, it deliver its correspondent batch as the decision for its respective consensus instance.
This is the message pattern that is executed if the leader is correct and the system is synchronous. If these conditions do not hold, the protocol needs to elect a new leader and force all replicas to converge to the same consensus execution. This mechanism is dubbed synchronization phase and is described in detail in [288].
As mentioned before, BFT-SMaRt can also be configured for crash fault tolerance (CFT). In this case it implements a Paxos-like message pattern [210], illustrated in Fig. 5.1b. The main differences are that the CFT protocol does not require WRITE messages, waits only for $\lceil \frac{n+1}{2} \rceil$ ACCEPT messages and requires a simple majority of non-faulty replicas to preserve correctness (the BFT protocol requires that more than three quarters of the replicas remain correct).



(a) BFT message pattern.

(b) CFT message pattern.

Figure 5.1: BFT-SMaRt message pattern during fault-free executions.

5.2 Experiments

In this section we present the experiments conducted to assess the effectiveness of certain optimizations proposed for SMR in wide area networks [210, 96, 229, 323, 304, 241, 197] and quorum systems [162, 262]. Before presenting our results, we describe some general aspects of our methodology.

5.2.1 Methodology

The considered optimizations were evaluated by implementing variants of the BFT-SMaRt's code and executing them simultaneously with the original protocol. Our experiments focus on measuring latency instead of throughput, in particular the median and 90th percentile latency perceived by clients. This is due to the fact that throughput can be effectively improved by adding more resources (CPU, memory, faster disks) to replicas or by using better links, whereas geo-replication latency will always be affected by the speed of light limit and perturbations caused by bandwidth sharing.
During the experiments, clients were equally distributed across all hosts, i.e., a BFT-SMaRt replica and a BFT-SMaRt client were deployed at each host that executed the protocol. Similarly to other works (e.g., [181, 241]), each client invokes 1kB-requests and receive 1kB-replies from the replicas, which run a null service. Requests were sent to the replicas every 2 seconds, and each client writes its observed latency into a log file. This setup enabled us to retrieve results that are gathered under similar network conditions without saturating either the CPU or memory of the hosts used.
The experiments in which we evaluated optimizations to the SMR protocol were conducted mostly in PlanetLab.[1] This testbed is known for displaying unpredictable latency spikes and highly loaded nodes [140]. These conditions allow us to evaluate the optimizations within unfavorable conditions.
Since our experiments are designed to evaluate solely the client latency in fault-free executions, we only report executions in which all hosts were online. However, since PlanetLab's host are regularly restarted and sometimes become unreachable, we could seldom execute each experiment during the same amount of time [140]. Therefore, we had to launch multiple executions for the same experiment, so that within each execution there would be a period in which all hosts were online. In any case, every experiment reported in this chapter considers at least 24 hours of measurements.
All experiments were configured to tolerate a single faulty replica. Each experiment was executed using between three to five hosts spread through Europe. The unavailability of nodes already mentioned led us to use a total of eight hosts through all experiments (see Table 5.1).
To validate our results in a global scale, two of the experiments described in the chapter were executed on Amazon EC2,[2] using *t1.micro* instances distributed among five different regions. We used the same methodology described for the PlanetLab experiments.

[1] `http://www.planet-lab.org`.
[2] `http://aws.amazon.com/ec2/`.

Country	City	Hostname
Poland	Wroclaw	planetlab1.ci.pwr.wroc.pl
England	London	planetlab-1.imperial.ac.uk
Spain	Madrid	planetlab2.dit.upm.es
Germany	Munich	planetlab2.lkn.ei.tum.de
Portugal	Aveiro	planet1.servers.ua.pt
Norway	Oslo	planetlab1.ifi.uio.no
France	Nancy	host4-plb.loria.fr
Finland	Helsinki	planetlab-1.research.netlab.hut.fi
Italy	Rome	planet-lab-node1.netgroup.uniroma2.it

Table 5.1: Hosts used in PlanetLab experiments

5.2.2 Number of Communication Steps

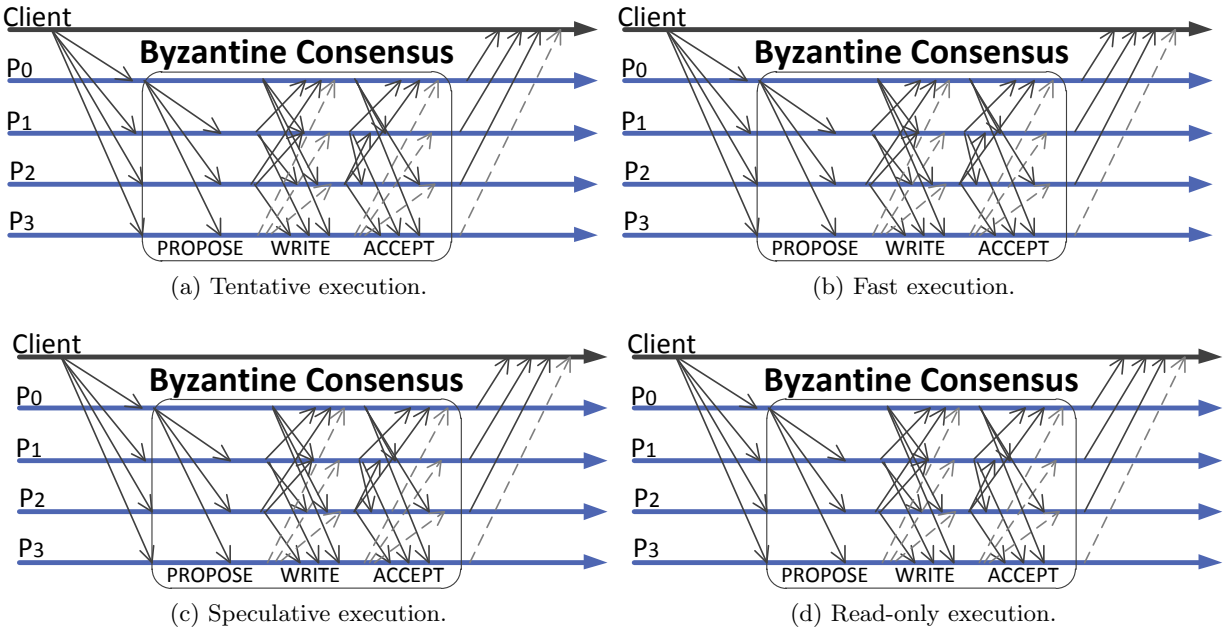
The purpose of our first experiment is to observe how the client latency is affected by the number of communication steps performed by the SMR protocol. More precisely, we wanted to observe how efficient read-only, tentative, speculative and fast executions are in a WAN. The first two optimizations are proposed in PBFT [96], whereas the other two optimization are used by Zyzzyva [197] and Paxos at War [323], respectively. Since these optimizations target Byzantine-resilient protocols, we only evaluate them in BFT mode.

The message pattern for each of these optimizations is illustrated in Fig. 5.2. Fig. 5.1a displays BFT-SMaRt's standard message pattern. Fig. 5.2a displays the message pattern for tentative executions. This optimization consists of delivering client requests right after finishing the WRITE phase, thus executing the ACCEPT phase asynchronously. This optimization comes at the cost of (1) potentially needing to perform a rollback on the application state if there is a leader change, and (2) forcing clients to wait for $\lceil \frac{n+f+1}{2} \rceil$ messages from replicas (instead of $f + 1$) [96]. Fig. 5.2b displays the message pattern for fast executions. This optimization consists of delivering client requests right after gathering $\lceil \frac{n+3f+1}{2} \rceil$ WRITE messages (before the ACCEPT phase finishes). If such amount of WRITE messages arrive fast enough, the protocol can safely bypass the ACCEPT phase. Fig. 5.2c displays the message pattern for speculative executions. This optimization enables the protocol to finish executions directly after the PROPOSE message is received in the replicas, as long as the clients are able to gather replies from all the replicas within a pre-established time window. If the clients are not able to gather all the replies within such time window, at least one additional round-trip message exchange is required to commit the requests. Fig. 5.2d displays the message pattern for read-only executions. This optimization enables clients to obtain a response from the service in two communication steps. However, it can only be used to read the state from the service. Similarly to tentative executions, this optimization also demands that clients gather $\lceil \frac{n+f+1}{2} \rceil$ messages from replicas, even for non-read-only operations, to ensure linearizability [96].

Setting: We created three BFT-SMaRt variants to evaluate fast, tentative and speculative executions (read-only executions were already supported). The replicas were deployed in Nancy (leader), Wroclaw, Helsinki and Rome.

Results: The values for the median and 90th percentile latency for each client are shown in Fig. 5.3. All evaluated optimizations exhibited latency reduction across all clients, with read-only executions finishing the protocol execution significantly faster than any of the other optimizations (i.e., 90th percentile latency from 43% to 63% smaller than in standard executions). Moreover, speculative executions also displayed significant latency reduction, reaching a 90th percentile latency 35% lower than standard execution. In the same way, tentative and fast executions also manage to reach a lower median and 90th percentile than standard executions, albeit with more modest differences. Furthermore, whereas fast executions displayed a latency decrease of about 10%, tentative executions managed to reduce latency by almost 20% (when compared to standard executions).

Main conclusion: The lowest latency displayed by read-only executions were to be expected, since they bypass all three communications steps executed between sending requests and gathering replies.



(a) Tentative execution.

(b) Fast execution.

(c) Speculative execution.

(d) Read-only execution.

Figure 5.2: Evaluated message patterns, besides the one in Fig. 5.1a.

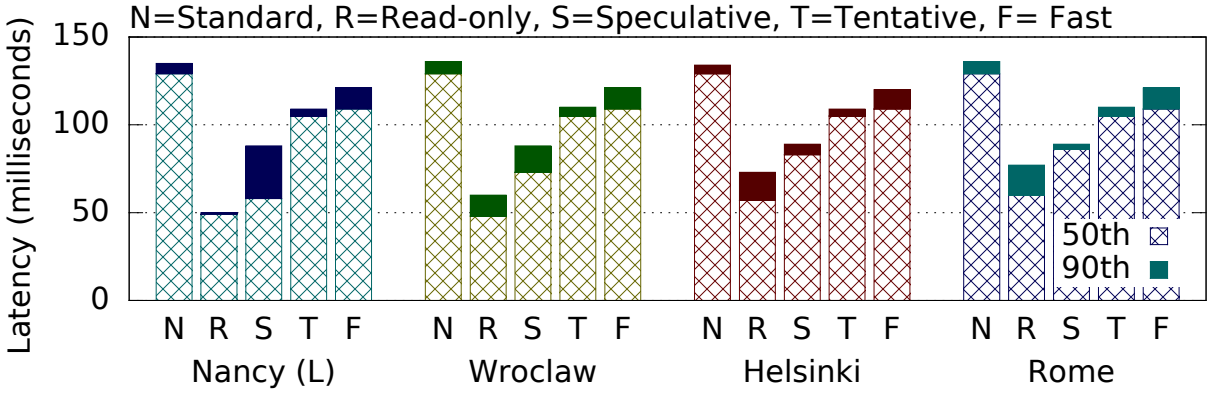


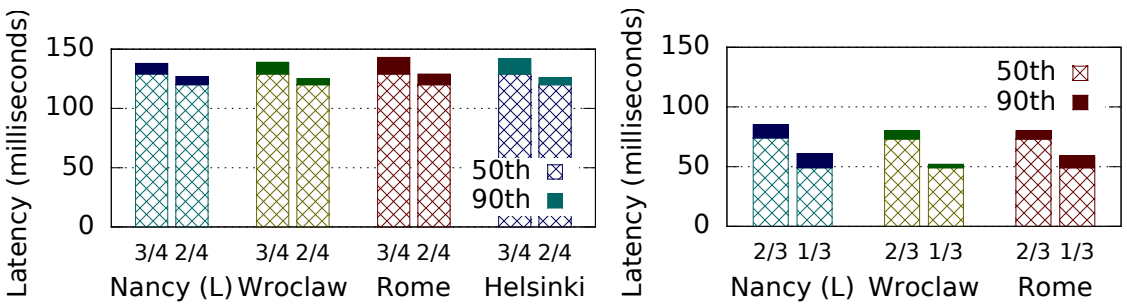
Figure 5.3: Client latencies' 50th/90th percentile for each type of execution.

Since speculative executions require the PROPOSE phase, they show higher latency than read-only executions. The advantage of tentative executions over fast executions can be explained by the fact that the latter require gathering WRITE messages from all four replicas, whereas the former only need it from three.

5.2.3 Number of Replies

In this experiment we intended to observe how the amount of replies required by clients affects the operation latency. By default, BFT-SMaRt clients wait for $\lceil \frac{n+f+1}{2} \rceil$ (BFT) or $\lceil \frac{n+1}{2} \rceil$ (CFT) replies from replicas in order to ensure linearizability. However, this number of replies is required due to the use of read-only executions [96]: if this optimization were not supported, $f+1$ matching replies (BFT) or 1 (CFT) replies would suffice.

Setting: We created a variant of a BFT-SMaRt client that waited only for $f+1$ (BFT) or 1 (CFT) replies, thus satisfying only sequential consistency (similarly to Zookeeper [181]) if the read-only optimization is employed. This experiment was deployed on hosts located in Nancy (leader), Wroclaw, Helsinki and Rome. The modified clients waited for two out of four replica replies (or one out of three in CFT), while the original version waited for the usual three out of four (two out of three



(a) BFT Mode.

(b) CFT Mode.

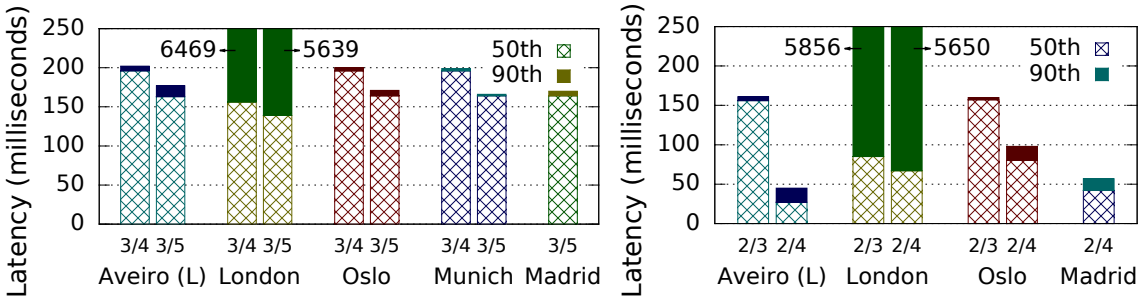
Figure 5.4: Client latencies' 50th/90th percentile for different numbers of replies.

in CFT). CFT experiments did not require the Helsinki's host.
Results: The values for the median and 90th percentile latency for each client are shown in Fig. 5.4. It can be observed that both the original and modified protocols present very similar performance in BFT mode. On the other hand, the optimization was quite effective in the CFT mode. For the 90th percentile, this optimization showed an improvement from 8% to 11% in BFT mode and from 26% to 36% in CFT mode.
Main conclusion: The lower latency displayed when the protocol requires less replies was to be expected, but such reduction was more significant in CFT mode. This can be explained by the fact that the BFT mode employed one more replica and required one more reply when compared to CFT.

5.2.4 Quorum Size

This experiment is motivated by the works of Gifford [162] and Pris [262], which use voting schemes with additional hosts to improve the availability of quorum protocols. As described in §5.1, BFT-SMaRt's clients and replicas always wait for $\lceil \frac{n+f+1}{2} \rceil$ messages from other replicas to advance to the next communication step (or $\lceil \frac{n+1}{2} \rceil$ in CFT mode). More precisely, BFT-SMaRt waits for dissemination Byzantine quorums [228] if operating in BFT mode and majority quorums [157] if operating in CFT mode. During this experiment, we enable the system to make progress without waiting for the aforementioned quorum types if spare replicas are present. Notice that this optimization, which is not employed in any SMR protocol, might lead to safety violations (discussed below).
Setting: We modified BFT-SMaRt to make replicas wait for only $2f + 1$ (resp. $f + 1$) messages in each phase of the BFT (resp. CFT) protocol, independently from the total number of replicas n.[3] This experiment was deployed on hosts located in Aveiro (leader), London, Oslo, Munich and Madrid. The original BFT-SMaRt was configured to execute across four replicas (three in CFT mode) and the modified version was configured to execute in five (four in CFT mode). The extra replica needed for executing the modified version was placed in Madrid, both for BFT and CFT mode. Experiments for CFT mode did not require the use of Munich's host. Since the modified version waits only for three out of five (3/5) messages (or 2/4 messages in CFT mode), both versions of BFT-SMaRt will wait for the same number of messages, even though the optimized versions use one additional replica.
Results: The values for the median and 90th percentile latency for each client are shown in Fig. 5.5. The results show that the modified protocols – which used one extra replica – exhibited lower latency than the original protocols. This difference is more discernible in the CFT mode for two reasons. First, the ratio between the quorum size and the number of replicas (2/4) is smaller than the BFT case (3/5). Second, it did not use London's host (which observed a much worse 90th percentile latency than others). It can be observed that in the 90th percentile, the optimizations showed an improvement of 12%-17% in the BFT mode and 4%-72% in CFT mode, depending on the location of clients.

[3] If the original BFT-SMaRt were deployed in five hosts, the quorums would be comprised of four hosts (in the case of BFT mode).



(a) BFT Mode.

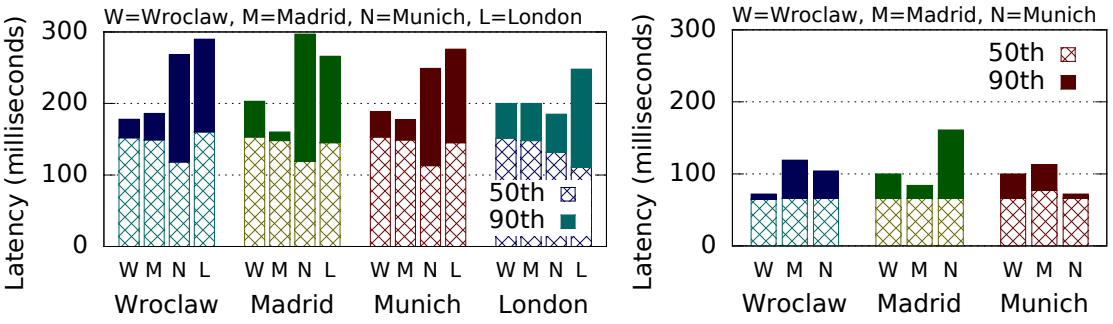
(b) CFT Mode.

Figure 5.5: Client latencies' 50th/90th percentile with different quorum sizes.

Main conclusion: The modified version BFT-SMaRt was able to experience lower latency because it was given more choice: since both versions still waited for the same number of messages in each communication step, the slowest replica was replaced by the extra replica hosted in Madrid, thus decreasing the observed latency of the modified version. In normal protocols this benefit would be smaller, since the quorum size would normally increase with n.
Even though the use of additional replicas decreases the protocol latency, this kind of optimization cannot be directly applied to existing protocols without impairing their correctness. Limiting the amount of messages to $2f+1$ (or $f+1$) regardless of the total number of replicas available n does not guarantee the formation of intersecting quorums, which are required to ensure safety in both BFT and CFT modes [210, 96]. For example, in the CFT mode, our setup of $n = 4$ and $f = 1$ did not ensure majority quorums, which could had lead to safety violations. In order to preserve correctness, it is necessary to force any combination of $2f+1$ (or $f+1$) replicas to intersect in at least one correct server. In §5.3.2 we present a technique that ensures this property and allows the use of this optimization in SMR systems.

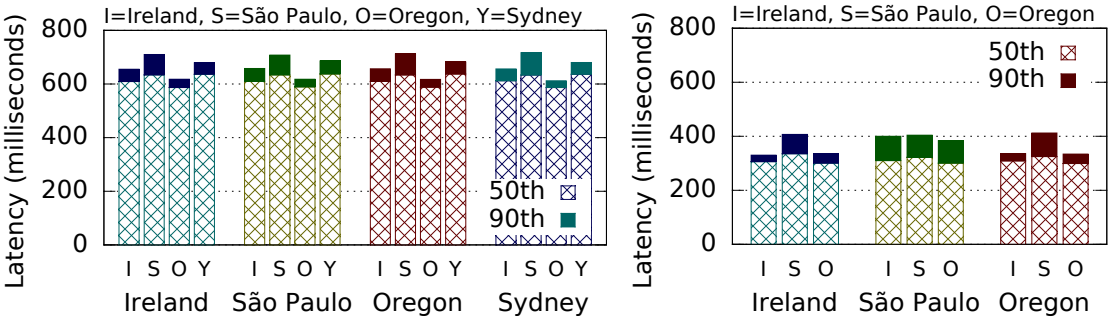
5.2.5 Leader Location

The goal of our last experiment is to observe how much the leader's location can affect the client latency. This experiment is motivated by the fact that Mencius [229], EBAWA [304] and EPaxos [241] use different techniques to make each client use its closest (or co-located) replica as the leader for its operations. The rationale behind these techniques is to make client-leader communication faster, bringing down the end-to-end SMR latency (see Fig. 5.1).
Setting: We deployed BFT-SMaRt in PlanetLab and conducted several experiments considering different replicas assuming the role of the leader. The hosts used were located in Wroclaw, Madrid, Munich and London (not used in CFT mode). Moreover, the experiment was repeated across Amazon EC2, using replicas in Ireland, Oregon, So Paulo and Sydney regions (Sydney was only used in BFT mode).
Results: Before launching this experiment, we expected that, for any client, its latency would be the lowest when its co-located replica were the protocol's leader. However, as seen in Fig. 5.6, the median and 90th percentiles of the latency observed by the different clients do not change significantly when the leader location changes. In particular, the 90th percentile latency is, in general, lower when the leader was either in Madrid or Wroclaw.
Since these results appeared to contradict the intuition of [229, 304, 241], we repeated this experiment in Amazon EC2, to find if this phenomenon is due to our choice of testbed. Fig. 5.7 shows the results observed in each Amazon EC2 region. As with the PlanetLab results, the latency observed by the different clients do not present any significant change as we change the leader location. However, having the leader in Oregon results in a slightly lower 90th percentile for all clients, both for BFT and CFT modes.



(a) BFT Mode. (b) CFT Mode.

Figure 5.6: Client latencies' 50th/90th percentile when the leader is placed across PlanetLab hosts.



(a) BFT Mode. (b) CFT Mode.

Figure 5.7: Client latencies' 50th/90th percentile when the leader is placed across Amazon EC2 regions.

Main conclusion: The obtained results depict a similar trend in the two different testbeds, which let us assert that co-locating clients with the leader does not necessarily improve the latency of replicated state machines. On the other hand, placing the leader in the host with better connectivity with the remaining replicas can yield more consistent improvements. More precisely, the benefit of reaching the leader faster is not as important as hosting the leader in the replica with faster links with others.

5.2.6 Discussion

The results presented in §5.2.2 indicate that, as expected, bypassing communication steps reduces client latency in BFT SMR protocols. However, even though read-only (resp. speculative) executions are up to 63% (resp. 35%) faster than standard executions, the benefits of tentative and fast executions are not so impressive: about 20% and 10%, respectively. The difference, as explained before, is due to the fact that fast executions requires larger quorums than tentative execution, which requires waiting for more messages (that can be slow in an heterogeneous environment such as a WAN). In the end, tentative execution matches the theoretically expected benefits: by avoiding 20% of the communication steps (see Fig. 5.2), we did reduce latency to approximately 20%.
The results of §5.2.3 and §5.2.4 show that decreasing the ratio between the number of expected messages and the total number of replicas can decrease latency significantly, especially for CFT replication. More specifically, clients that wait less replies had a 90th percentile latency improvement of up to 36% (resp. 11%) in CFT (resp. BFT) mode; and adding more replicas to the system while maintaining the same quorum size brings improvements of up to 72% (resp. 17%) in CFT (resp. BFT) mode. These results are mainly due to the performance-heterogeneity of hosts and links in real wide area networks: if the latency between all replicas were similar and network delivery variance were small, the observed improvements would be much more modest. Furthermore, they are in accordance with other

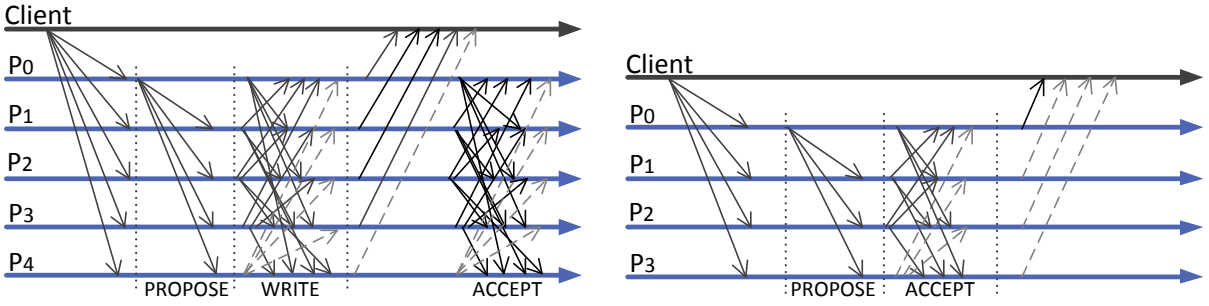
studies showing that using smaller quorums may bring better latency than decreasing the number of communication steps (e.g., [188]).
The results of §5.2.5 indicates that having the leader close to a client will not significantly reduce the SMR latency for this client. This result is unexpected since several protocols implement mechanisms such as rotating coordinator [229, 304] and multiple proposers [241] to make each client submit its requests to the closest replica. We found two main explanations for this apparent contradiction. First, the heterogeneity of real environments such as PlanetLab and Amazon EC2 make optimizations for reducing latency less effective. In fact, the authors of Mencius acknowledge that the protocol achieves lower latency than Paxos only in networks with small latency variances [229]. Second, in CFT mode, BFT-SMaRt clients wait for replies from a majority of replicas to ensure linearizability due to the use of the read-only optimization. EPaxos, Mencius and Paxos clients wait only for a single reply from the leader. This means that client-leader co-location in these protocols potentially reduce the latency in two communication steps, while in BFT-SMaRt this reduction is in only one (clients still need to wait for at least one additional reply). Consequently, having a client co-located with the leader should decrease the number of communication steps 25% in CFT mode and 20% in BFT mode, while in Mencius and EPaxos such theoretical improvement can reach 50%. Moreover, its worth to point out that these benefits appear only in favorable conditions. For example, EPaxos presents almost the same latency of Paxos when under high request interference [241].
As a final remark, it is worth noting that our results show that having a leader in a well-connected replica brings, in general, more benefits than having clients co-located with leaders. For instance, we observed that latency was usually lower when the leader replica was hosted in Madrid, rather than when the leader replica was placed in the same location as a particular client. In the same line, adding faster replicas to the system may significantly improve latency, as shown in §5.2.4. For example, the addition of Madrid to the set of replicas decreased the 90th percentile latency in Oslo and Aveiro by 39% and 72%, respectively (CFT mode). More generally, these results highlight the fact that not all replicas are the same in geo-replication and that both the leader location and quorum formation rules must take into account the characteristics of the sites being used.

5.3 The WHEAT Protocol

This section describes WHEAT, a WAN-optimized SMR protocol implemented on top of BFT-SMaRt. We start by discussing the WAN optimizations employed in the protocol and then introduce two novel vote assignment schemes to use smaller quorums without endangering the safety of SMR. We conclude the section with an evaluation of WHEAT in Amazon EC2.

5.3.1 Deriving the protocol

WHEAT employs the optimizations that were most effective in improving the latency of SMR in WANs. The selected optimizations (discussed below) reduce the number of communication steps, the number of replies that clients wait and the ratio between the quorum size and the total number of replicas. Since the results of client-leader co-location were not so expressive, and given that its implementation would require substantial changes in the base SMR protocol (which is already complex enough [68, 100]), we rejected this optimization and followed the fixed leader approach. As with BFT-SMaRt, WHEAT can be used in BFT or CFT modes, implementing the message patterns illustrated in Fig. 5.8.
Reducing the number of communication steps: In BFT mode, WHEAT employs the read-only and tentative execution optimizations introduced in PBFT [96]. The reason to support tentative executions instead of fast or speculative executions is as follows: (1) during our experiments, tentative executions displayed slightly better latency than fast executions (i.e., they had a lower 90th percentile); (2) speculative executions are useful in environments were the network is predictable and stable, which we cannot expect in many geo-distributed settings. If such conditions are not met by the network (i.e, not delivering replies from all replicas within the required time window), clients need to trigger the



(a) BFT message pattern.

(b) CFT message pattern.

Figure 5.8: WHEAT's message pattern for $f = 1$ and one additional replica.

commit phase and force the protocol to execute five communications steps [197]; and (3) tentative executions do not require modifications to the synchronization phase of BFT-SMaRt. Fast executions would required modifications to account for cases where a value was decided solely with $\lceil \frac{n+3f+1}{2} \rceil$ WRITE messages, whereas the rollback operation can be triggered using the state transfer protocol already implemented in BFT-SMaRt [68]. Furthermore, usage of speculative executions would demand the complete re-implementation of the original protocol, to account for the several corner cases necessary to preserve correctness under this type of executions, such as the aforementioned commit phase. Another advantage of tentative executions is that ACCEPT messages can be piggybacked in the next PROPOSE or WRITE messages, similarly to PBFT[96].

Reducing the number of replies a client waits: In BFT mode, the use of read-only and tentative executions lead WHEAT clients to always gather responses from a Byzantine quorum of replicas, i.e., at least $\lceil \frac{n+f+1}{2} \rceil$ replies. This means that it is impossible to enforce the optimization evaluated in §5.2.3 without giving up linearizability [175]. However, single-reply read-only executions can still be used in the CFT mode as long as clients always contact the leader replica.[4] Consequently, in CFT mode WHEAT clients only need to wait for one reply (from any replica during write operations and from the leader during read-only operations).

Reducing the ratio between the quorum size and the number of replicas: As observed in §5.2.4, it is possible to significantly decrease latency by adding more replicas to the system, as long as the quorums used in the protocol remain with the same size. Both the Byzantine and crash variants of WHEAT are designed to exploit this phenomenon by modifying the quorum requirements of the protocol. However, to avoid breaking the safety properties of traditional SMR protocols (e.g., [210, 96, 304]), we need to introduce a mechanism to secure the formation of intersecting quorums of variable size. In the next section we introduce a voting scheme that preserves this requirement.

5.3.2 Vote Assignment Schemes

Our novel voting assignment schemes integrate the classical ideas of weighted replication [157, 162, 262] to SMR protocols. The goal is to extend quorum-based SMR protocols to (1) rely primarily on the fastest replicas present in the system, and (2) preserve its original safety and liveness properties.

The most important guarantees that quorum-based protocols need to preserve are (1) all possible quorums overlap in some correct replica and (2) even with up to f failed replicas, there is always some quorum available in the system. In CFT protocols like Paxos [210], quorums must overlap in at least one replica. Such intersection is enforced by accessing a simple majority of replicas during each communication step of a protocol. More specifically, protocols access $\lceil \frac{n+1}{2} \rceil$ replicas out of $n \geq 2f+1$. BFT protocols like PBFT [96], on the other hand, usually employ disseminating Byzantine quorums [228] with at least $f + 1$ replicas in the intersection. In this case, protocols access $\lceil \frac{n+f+1}{2} \rceil$ replicas out of $n \geq 3f + 1$. With this strategy, adding a single extra replica to the system results

[4] It is also necessary to use leases on the client, since the leader can be demoted at any point.

in higher latency, since any possible quorum becomes larger in size – unlike the weighted quorums strategy we present below.

The fundamental observation we make is that accessing a majority of replicas guarantees the aforementioned intersection, but this is not the only way to secure such intersection. More specifically, if n is greater than $2f + 1$ (in CFT mode), it is possible to distribute weights across replicas in such way that a majority is not always required to (correctly) make progress. As an example, consider the quorums illustrated in Fig. 5.9 (with one extra replica in the system). Whereas in Fig. 5.9a the intersection is obtained by strictly accessing a majority of replicas, in Fig. 5.9b we see that we can still obtain an intersection with a variable number of replicas (since we can obtain a sum of 3 votes by either accessing 2 or 3 replicas). In particular, if the replica with weight 2 is successfully probed, the protocol can finish a communication step with a quorum comprised by only half of the replicas. Otherwise, a quorum comprised by all replicas with weight 1 is necessary to make progress. Notice that for this distribution to be effective, it is necessary to attribute weight 2 to the fastest replica in the system.

We now generalize the weight distribution proposed in Fig. 5.9b to account for other values of f. The objective is to assign certain numbers of votes (i.e., weights) to each replica in accordance with their connectivity/performance. This vote assignment must be done carefully to ensure that minimal quorums composed by faster replicas will be used under normal conditions (i.e., when the faster replicas are indeed faster) and larger, yet available quorums can be used to ensure that up to f faulty replicas are tolerated (despite their weights).

Let Q_v be the minimum number of votes that a quorum of replicas must hold to guarantee that quorums overlap by at least one correct replica. A quorum is said to be safe and minimal (or just minimal) if it is comprised by only $f + 1$ replicas that together hold Q_v votes. This quorum size is minimal because if f or less replicas were considered a quorum, other intersecting quorums would require more than $n - f$ replicas. These quorums will not be available when there are f faulty replicas in the system. This means that having quorums with less than $f + 1$ replicas implies giving up consistency or availability, as described in classical quorum definitions [228]. In a BFT system, for the same reasons, a minimal quorum must be comprised of $2f + 1$ replicas.

Using the above definitions, we consider vote distribution schemes that satisfy the following properties:

- **Safe minimality:** There exists at least one minimal quorum in the system.
- **Availability:** There is always a quorum available in the system that holds Q_v votes.
- **Consistency:** All quorums that hold Q_v votes intersect by at least one correct replica.

In the following, we describe vote assignment schemes for CFT and BFT modes that satisfy these properties.

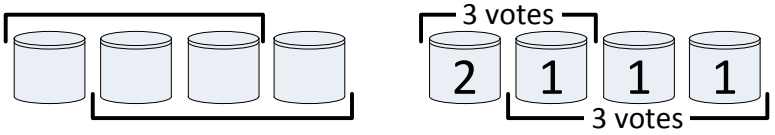
CFT vote distribution: To calculate the vote distribution under CFT mode, we start by introducing the parameter Δ, which represents the number of extra replicas available in the system. Thus, n can be calculated using Δ as follows:

$$n = 2f + 1 + \Delta \tag{5.1}$$

We now introduce two additional variables. N_v represents the sum of the number of votes $\sum V_i$ that are attributed to each replica i. F_v is the maximum number of votes that can be dismissed in the system. Having these parameters, we can apply the standard quorum rules to the votes instead of the replicas. Hence, N_v is calculated as follows:

$$N_v = \sum V_i = 2F_v + 1 \tag{5.2}$$

As an example, consider Fig. 5.9b: the sum of all votes adds up to 5, which represents an abstract quorum system comprised by 5 hosts capable of withstanding 2 faults. Therefore, for this case, $N_v = 5$ and $F_v = 2$.



(a) Classical (always a majority of replicas).

(b) Weighted (from $f+1$ to $n-f$ replicas).

Figure 5.9: Quorum formation when $f = 1$ and $n = 4$ (CFT mode).

Since Δ and f are the input parameters, we need to (1) find a relation between Δ and f and values for N_v, F_v and V_i; (2) use those variables to force the emergence of replica quorums that intersect by one replica. More precisely, votes must be distributed in such a way that once $Q_v = F_v + 1$ votes are gathered, quorums always overlap by at least one correct replica.
If we assume that only two possible values can be assigned to replicas (e.g., a binary vote distribution), as in Fig. 5.9b, we can introduce variables V_{max} and V_{min}. However, we need to find how many replicas are assigned V_{max} and V_{min}. Let u be the number of replicas holding V_{max} votes and, consequently, $n - u$ the number of replicas holding V_{min}. Since the sum of all votes must be equal to N_v, we have:

$$N_v = 2F_v + 1 = uV_{max} + (n-u)V_{min} \tag{5.3}$$

In the example of Fig. 5.9b, $V_{max} = 2$, $V_{min} = 1$ and $Q_v = F_v + 1 = 3$. We can observe two cases where 3 votes can be obtained: either by (1) accessing the single V_{max} replica and one of the V_{min} replicas, or (2) accessing all V_{min} replicas. Notice that in both cases, the same number of votes is dismissed, but not the same number of replicas; in case (1) two replicas are ignored, but in case (2) only one single replica is left unprobed. Also note that the number of votes dismissed is 2 – which happens to be the value of F_v (as we pointed out previously). This indicates that F_v has a direct relation to V_{max} and V_{min}. Given this observation, we generalize this example scenario to represent any Δ and f:

$$F_v = (\Delta + f)V_{min} = fV_{max} \tag{5.4}$$

We derive the relation between V_{max} and V_{min} as follows:

$$V_{max} = \frac{(\Delta + f)}{f} V_{min} \tag{5.5}$$

If we assume $V_{min} = 1$, equations 5.4 and 5.5 become:

$$F_v = \Delta + f \tag{5.6}$$

$$V_{max} = \frac{\Delta + f}{f} = 1 + \frac{\Delta}{f} \tag{5.7}$$

Having now more refined formulas for F_v, V_{max} and V_{min}, we can return to equation 5.3 and obtain the value of u:

$$2(\Delta + f) + 1 = u(1 + \frac{\Delta}{f}) + (n - u) \Rightarrow u = f \tag{5.8}$$

Knowing that $u = f$, still by equation 5.3, there must be f replicas holding V_{max} votes and $n - f$ replicas holding 1 vote (since $V_{min} = 1$). We thus have our CFT vote assignment scheme: equations 5.7 and 5.8 give us the values for F_v and V_{max} respectively, all in function of Δ and f.
The main benefit of this scheme is that if all the f replicas holding V_{max} are probed faster than any other, then just one of the $\Delta + f + 1$ other replicas holding V_{min} votes will be disregarded (like the two-replica quorum of Fig. 5.9b). However, in the worst case, if f replicas holding V_{max} votes fail (or are slow), then all replicas with V_{min} votes will be accessed instead (as the three-replica quorum of Fig. 5.9b).
CFT proof of correctness: In the following we briefly outline the proof that our vote assignment scheme satisfies the three properties described before. The full proof is available in the extended technical report [287].

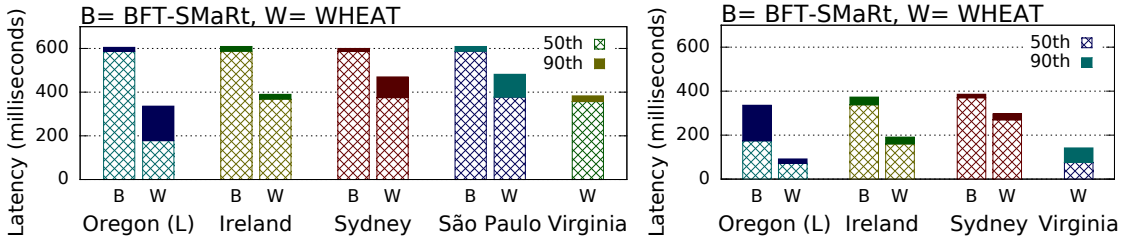
Safe minimality: Let S_{max} to be the subset of f replicas that hold V_{max} votes each. By equation 5.4 these f replicas will add up to F_v votes. A safe and minimal quorum can be built using S_{max} plus one additional replica holding $V_{min} = 1$ votes, making a quorum with $Q_v = F_v + 1$ votes. □
Availability: Let S_{min} be the subset of $n - f$ replicas holding V_{min} votes each. In the worst case (maximum number of votes lost due to faults), when all the f replicas holding V_{max} fail, there will be still the $n - f = \Delta + f + 1$ replicas from S_{min}. According to equation 5.7, $\Delta + f$ account already for F_v votes. With the additional replica from S_{min}, we reach the required $Q_v = F_v + 1$ votes to form a quorum, even with the fV_{max} votes lost. Furthermore, any other combination of V_{max} and V_{min} replicas will always contain at least Q_v votes (as long there are at least $f + 1$ replicas), since they will either be a minimal quorum (as proved before), a S_{min} quorum, or a hybrid of both (which will result in $Q_v \geq F_v + 1$). □
Consistency: Any quorum overlaps by at least one correct replica for the following reason: since a minimal quorum contains Q_v votes (as proved before), it must contain at least one V_{min} replica, which in turn must be a member of the S_{min} quorum (which also contains Q_v votes, as proved before). Since any other allowed combination of V_{max} and V_{min} replicas will either be a superset of a minimal or S_{min} quorum, they will also intersect by one replica (or more). □
BFT assignment: The reasoning here is similar to the CFT scheme, but with the following differences. First, equations 5.1 and 5.2 become $n = 3f + 1 + \Delta$ and $N_v = \sum V_i = 3F_v + 1$, respectively. These equations still lead to the same values of F_v and V_{max}, but u becomes $2f$ instead of f. This forces the system to have $2f$ replicas holding V_{max} and $\Delta + f + 1$ replicas holding one vote (V_{min}). Moreover, it is necessary to gather $2F_v + 1$ votes on each quorum, which makes $Q_v = 2F_v + 1$. Finally, a minimal quorum must be comprised by $2f + 1$ replicas instead of $f + 1$. A complete description of the BFT voting assignment scheme and its correctness proof can be found in the extended technical report [287].
Additional resilience benefits: Besides the performance benefits, our voting assignment schemes present three benefits in terms of resilience. First, it allows the system to tolerate more than f crash faults in certain scenarios. For instance, in Fig. 5.9b, two of the V_{min} replicas could fail by crash and the protocol would still make progress without violating safety. However, this is not the case if one of two failed replicas holds V_{max} votes. Second, our assignment schemes could be used to assign a higher number of votes to replicas on more reliable and available sites (instead of the faster ones), improving thus the reliability and availability of the system. Third, when any of the faster replicas is detected as slow or unavailable, BFT-SMaRt's reconfiguration protocol [68] can be used to redistribute votes, so that other replicas take the place of the ones that are no longer the fastest. Notice that this approach is better than using BFT-SMaRt's reconfiguration protocol to replace unavailable replicas. Such replacement would require a state transfer, which can be a slow operation for large state sizes and limited wide-area links. For example, a 4GB-state will take more than fifty minutes to be transferred in a 10 Mbps network (which is better than most links between EC2 regions). With our approach, the extra replicas are already active and up-to-date in the system, so the reconfiguration takes approximately the time to execute a "normal" SMR operation. Finally, it is worth mentioning that in the event that the systems experiences a period of high load, it is possible that the minimal quorum becomes overloaded and unable to reply faster than other quorums, thus forcing the system to make progress with different quorums. Nonetheless, any SMR protocol based on quorum systems is subject to this issue.

5.4 Implementation and Evaluation

We implemented WHEAT by extending BFT-SMaRt for supporting the chosen optimizations (§5.3.1) and considering replicas with different number of votes (§5.3.2). Most of the modifications to the code took into account the vote assignment schemes that calculate the quorums used in the protocol.
We evaluated WHEAT by running a set of experiments in Amazon EC2 and comparing the results with the original BFT-SMaRt system. As in the EC2 experiments reported in §5.2.5, we use sites on

Sites	Ireland	So Paulo	Oregon	Sydney	Virginia
Ireland	0	211 ± 10	171 ± 11	340 ± 11	88 ± 10
So Paulo	208 ± 14	0	217 ± 19	359 ± 4	123 ± 3
Oregon	171 ± 14	217 ± 11	0	205 ± 7	70 ± 12
Sydney	336 ± 26	359 ± 4	205 ± 10	0	255 ± 12
Virginia	88 ± 10	123 ± 4	71 ± 13	256 ± 5	0

Table 5.2: Average roundtrip latency and standard deviation (milliseconds) between Amazon EC2 regions as measured during a 24 hour-period.



(a) BFT Mode.

(b) CFT Mode.

Figure 5.10: 50th/90th percentile latencies observed by BFT-SMaRt and WHEAT clients in different regions of Amazon EC2.

Ireland, Oregon, Sydney and So Paulo (only in BFT mode) for BFT-SMaRt using also Virginia as the additional replica of WHEAT. This means that the original version of BFT-SMaRt employed 4 replicas in BFT mode (resp. 3 in CFT mode) whereas WHEAT employed 5 replicas in BFT mode (resp. 4 in CFT mode), with two of these replicas in North America. In BFT mode, the following parameters were employed (obtained through the voting schemes described previously): $N_v = 7$, $F_v = 2$, $V_{max} = 2$ for the replicas in Oregon and Virginia. In CFT mode, the configuration was $N_v = 5$, $F_v = 2$, $V_{max} = 2$ for the replica in Virginia. We attributed the V_{max} values to the these sites because they were the ones with better connectivity to others, as shown in Table 5.2.

The median and 90th percentile latencies for each client location and protocol is presented in Fig. 5.10. By employing the selected optimizations (§5.3.1) and using an additional replica in Virginia without increasing the quorum requirements (i.e., three and two replicas for BFT and CFT, respectively), WHEAT achieves, when compared to BFT-SMaRt, a 90th percentile latency improvement between 21% and 44% (BFT) and between 23% and 73% (CFT). Interestingly, the client in the leader region (Oregon) observed significant improvements, with median latency values matching the roundtrip times between Oregon and Ireland (BFT mode) or Virginia (CFT mode). This is a consequence of the fact that this client is co-located with the leader in the most well-connected site of the system. Moreover, upon considering all clients' measurements together, we found that WHEAT improved the global 90th percentile by 35% (BFT) and 28% (CFT). The global median improvement is even higher: 37% in BFT and 56% in CFT.

The improvements shown in this experiment should be taken with a bit of salt since they may be due the use of an additional site with a good roundtrip latency with other replicas (see Table 5.2). If the new replica used in WHEAT were added on an hypothetical Amazon EC2 region "moon" (instead of Virginia), with a higher roundtrip latency with all other sites, the WHEAT results would be less impressive since the faster quorums will be the same of BFT-SMaRt. The only benefits will be due to the other optimizations (tentative executions for BFT and single-reply for CFT) implemented in the system. Nonetheless, our results illustrate the fact that in a real geo-replication setup there are significant benefits in assigning different weights to different replicas. Furthermore, even with the required algorithmic support, it is important to choose the location of the spare replicas employed in WHEAT, to ensure the minimal quorums will bring significant benefits.

A note on throughput: WHEAT aims to improve geo-replication latency, and thus all of its

optimizations target this performance metric. However, the fact it uses Δ more replicas than BFT-SMaRt, implies it might achieve a slight lower peak throughput than the original system. This happens because more replicas lead to more message transmissions, which results in higher CPU and network bandwidth utilization. More precisely, each consensus instance on BFT-SMaRt requires the exchange of $3f + 18f^2$ (resp. $2f + 4f^2$) messages in BFT mode (resp. CFT mode), whereas in WHEAT it requires $3f + \Delta + 2(3f + \Delta)^2$ (resp. $2f + \Delta + (2f + \Delta)^2$) message exchanges. Although undesirable, this drawback will only affect a saturated system, which is rarely the case in production environments. Moreover, as discussed in §5.2.1, throughput can be improved by increasing CPU and network resources, while latency can only be addressed by better protocols.

5.5 Related work

The criticality of modern internet-scale services have created the need for geo-replication protocols for disaster tolerance, including whole-datacenter failures (e.g., [117]). Following this trend, several works proposed strongly consistent WAN SMR protocols [35, 229, 304, 241].

Steward [35] is a hierarchical Byzantine fault-tolerant protocol for geographically dispersed multi-site systems. It employs a hybrid algorithm that runs a BFT agreement protocol within each site, and a lightweight, crash fault-tolerant protocol across sites. Even though Steward is able to perform well in WANs (when compared with PBFT [96]), that comes at the cost of a very complex protocol (over ten specialized algorithms that run within and among sites) that demands plenty of resources (e.g., each site requires at least 4 replicas). Although we advocate the use of additional replicas for improving latency in WHEAT, our protocol is not radically different from "normal" protocols, requiring no specialized subprotocols or a specific number of replicas on a site.

Mencius [229] is an SMR protocol derived from Paxos [210] also optimized to execute in WANs. Like Paxos, it can survive up to f crashed replicas out of at least $2f + 1$. 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 become the leader. EBAWA [304] is a Byzantine-resilient SMR protocol optimized for WANs. It considers a hybrid fault model in which each replica uses a local trusted/trustworthy service (that cannot be compromised) to provide tolerance to up to f Byzantine faults using only $2f + 1$ replicas. Similarly to Mencius, it uses a rotating leader to allow clients to send requests to the replicas that are close to them. Egalitarian Paxos (EPaxos) [241] is a recent SMR protocol also derived from Paxos and designed to execute in WANs. Unlike most SMR protocols inspired by Paxos, EPaxos does not rely on a single designated leader for ordering operations. Instead, it enables clients to choose which replica should propose their operations, and employs a mechanism for solving conflicts between interfering operations. Differently from Mencius, EBAWA and EPaxos, WHEAT does not employ any mechanism to make clients use their closer replicas as leaders/coordinators/proposers. Our decision to avoid this optimization comes from observing that having a leader in the same site as the client gives less benefits in terms of latency than using the fastest replica as the leader.

Weighted replication was originally proposed by Gifford [162], and then revisited by Garcia-Molina [157] and Pris [262]. While Gifford made all hosts hold a copy of the state with distinct voting weights, Pris made 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). More recent works confirmed the usefulness of these ideas also for performance by showing that adding few servers to a group of replicas can significantly improve the access latency of majority quorums [51], and the same kind of technique is being used in practical systems to improve tolerance to slow servers [129]. By contrast, Garcia-Molina addresses the idea of weighted replication in [157] for coterie systems, which later evolved into the classic quorum systems without including vote distribution. Unlike our approach, none of these works target geo-replication: [262] and [157] are strictly theoretical contributions and [162] considers a local datacenter. To the best of our knowledge, we present the first vote assignment scheme that unpacks a weight distribution in function of the expected number of faults

and the amount of spare replicas available in the system.
There are empirical studies which evaluate the availability of quorum systems (e.g., [36, 51]), the latency of distributed algorithms over the internet (e.g., [50]) and the performance of different total order broadcast protocols – a fundamental building block for SMR – over a WAN (e.g., [40, 142, 277]). Our experiments have a different goal: instead of evaluating the performance of distinct protocols, we compare geo-replication-related optimizations employed by different protocols, but implemented in the same codebase, to validate the effectiveness of these optimizations in real WANs.

5.6 Conclusion

In this chapter we revisited some optimizations proposed in the literature for improving the latency of SMR protocols in wide area networks. More concretely, we implemented such optimizations in an open-source SMR library and compared its latency with a non-optimized version in the PlanetLab testbed and Amazon EC2 cloud to assess which of these optimizations bring significant benefits. Our results indicated that removing communication steps and demanding less replies from replicas lead to latency reductions of up to 20%, depending on the hosts and fault model. Surprisingly, using the closer replica as the leader held less benefits than what was expected. These results guided our design for WHEAT, an SMR protocol optimized for geo-replication that can be configured either for crash-only or Byzantine fault tolerance. WHEAT was implemented by extending BFT-SMaRt with the optimizations we observed as most effective and implementing novel vote assignment schemes for efficient quorum usage. Our evaluation showed gains of up to 56% for certain configurations, when compared with the unmodified BFT-SMaRt.

Chapter 6 Elastic State Machine Replication

As discussed in previous chapters, State Machine Replication (SMR) is a well-known approach to replicate a service for fault tolerance [278]. The key idea is to make replicas deterministically execute the same sequence of requests in such a way that, despite the failure of a fraction of the replicas, the remaining ones have the same state and ensure the availability of the system. Many production systems use this approach to tolerate crash faults [7, 11, 88, 79, 100, 117, 181], mostly by implementing Paxos [210] or a similar algorithm [189, 248, 250].

A critical limitation of the basic SMR approach is its lack of scalability: (1) the services usually need to be single-threaded to ensure replica determinism [278], (2) there is normally a leader replica which is the bottleneck of the SMR ordering protocol, and (3) adding more replicas does not improve the system performance. Different techniques have been developed to deal with these limitations. Some works propose SMR implementations that take advantage of multiple cores to execute requests in parallel [198, 169, 192], solving (1). Although effective, the improvements are limited by the number of cores available on servers. In the same way, some unorthodox protocols spread the additional load imposed to the leader among all the system replicas [229, 241]. These protocols solve (2), but scalability remains limited because every replica still needs to execute all the operations.

A recent line of work proposes partitioning of the SMR-based systems in multiple Replicated State Machines (RSMs) [163, 253, 117, 258] for addressing (3). Although partitioning solves the scalability of SMR, the existing solutions are quite limited in terms of elasticity, i.e., the capacity to dynamically increase (scale-out) and decrease (scale-in) the number of partitions at runtime. More specifically, some scalable systems consider only static partitions [258, 253], with no elasticity at all, while others [163, 117] provide dynamic partitioning through ad-hoc protocols that are executed in the background to avoid performance disruption, but with negative implications on the time needed to complete the partitioning.

This chapter introduces a generic partition transfer primitive (and protocol) that enables SMR-based services to be elastic and more fitted to the Cloud environment. The proposed protocol is designed to perform partition transfers efficiently and with minimal perturbations on the client-perceived performance on top of any existing SMR protocol.

Although SMR was traditionally employed for building metadata and coordination services (e.g., [88, 79, 181]), recent works have been pushing the use of this technique for implementing high-performance storage services [7, 11, 117, 67, 70, 253, 268, 313]. Elastic SMR gives these services the ability to increase and decrease their capacity in terms of storage and throughput. Figure 6.1 illustrates three situations in which such elasticity might be beneficial. A group G is used up to its maximum load capacity and, then, the state managed by this group is split to another group L to balance the load among the two groups and open room for the system to handle more work. When two groups are processing requests, a non-uniform access pattern might unbalance the load handled by the two groups, as shown in the second case. Here, part of the state of the overloaded group G can be transferred to the underloaded group L to rebalance the system. Finally, if the load decreases and one of the partitions becomes underutilized, it is possible to merge the state of the two groups in a single RSM, for freeing the underutilized resources.

An important issue in elastic systems is to define when and how to perform reconfigurations like the ones shown in Figure 6.1. There are many works on dynamic resource configuration managers [202, 246, 155], which decide when and how to adapt according to specified policies. A fundamental parameter

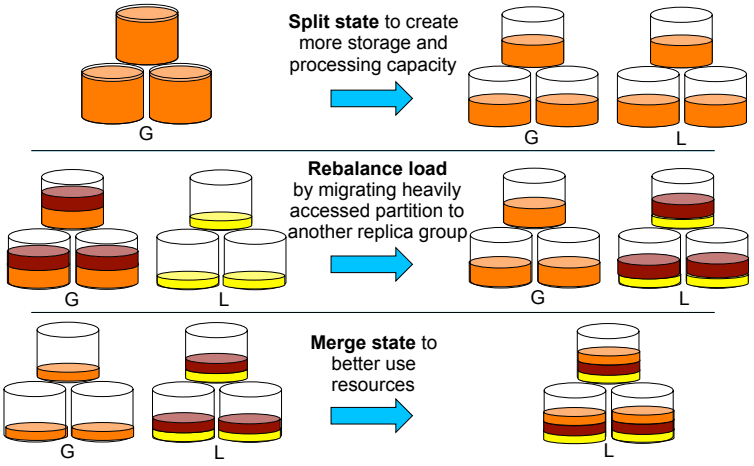


Figure 6.1: Reconfigurations of a partitionable RSM.

used in these systems is the amount of time required for deploying a new server and make it ready to process requests, i.e., the setup cost. A high setup cost always discourages reconfigurations, leading to over-provisioning and increased operational costs [155]. Given that, it is extremely important to reduce the setup cost.

In contrast to a stateless service, where servers start processing requests as soon as they are launched, in a stateful service a partition transfer must be performed before a server starts processing requests. As a result, the setup cost increases. Therefore, tackling the cost of performing a partition transfer is crucial when stateful services are deployed on cloud systems in order to meet the requirements defined in Service Level Agreements (SLA). Usually, elastic stateful systems rely on distributed cache layers [247], write-offloading [244] and, ultimately, over-provisioning [155]. These techniques are useful to create buffers to either absorb unexpected load spikes or buy time for the stateful backend to reconfigure. In this work we challenge this design, at least for an important class of systems (SMR- or Paxos-based systems), and define a principled way for replicated stateful systems to scale-in and -out efficiently, for non-negligible partition sizes and workloads.

We implement our partition transfer protocol in an existing state machine replication programming library, and build a strongly-consistent elastic key-value store on top of it. We evaluate this system by performing several scaling operations, measuring the duration of partition transfers and the impact of these reconfigurations on the latency and throughput of the system. Our results show that the proposed solution effectively supports fast reconfigurations, allowing the system to quickly adapt to changes in its workload, something that is not achievable in current elastic (stateful) systems [116, 121, 196].

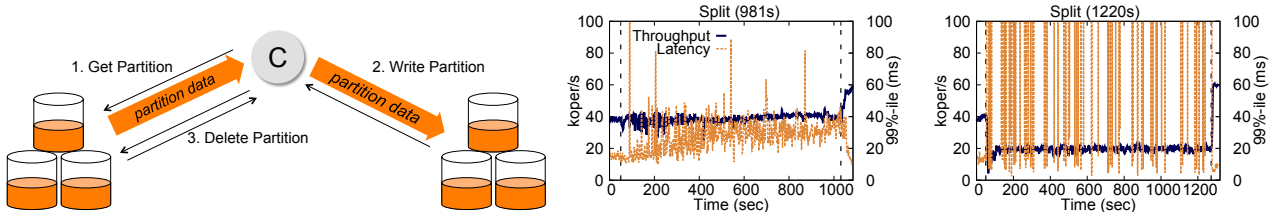
In summary, the contributions of this work are:

1. Surveys how elasticity is (or can be) implemented in existing SMR-based services and experimentally demonstrates their inefficiencies (§6.1);

2. Introduces a modular partition transfer primitive and protocol that enables SMR-based services to efficiently split and merge its state (§6.2);

3. Describes an implementation of this primitive in an open-source SMR library (§6.3);

4. Provides an extensive evaluation of the partition transfer protocol assessing its impact on key SLA-related metrics such as the duration of the reconfiguration process, the service throughput and the operation latency (§6.4).



(a) Client-based partition transfer. (b) Blocks of 4MB. (c) Blocks of 16MB.

Figure 6.2: A client-based protocol for partition transfer between RSMs with throughput and operation latency observed by clients during a 4GB-partition transfer operation (in multiple blocks of 4MB and 16MB).

6.1 Elasticity for RSMs

Dividing a RSM in multiple independent shards allows the system to scale linearly with the number of servers (as long as most operations affect a single partition). Most works following this approach assume statically defined partitions. The few that do support dynamic state partitioning [117, 163], neither specify how the partition is transferred to the new set of replicas nor how to make such transference as fast as possible and with minimal performance interference on the system. Having a partition transfer primitive for executing the operations illustrated in Figure 6.1 will allow SMR-based services to grow and shrink with the demand, thus supporting elasticity. In this section we discuss some potential solutions for implementing such primitive and discuss their limitations.

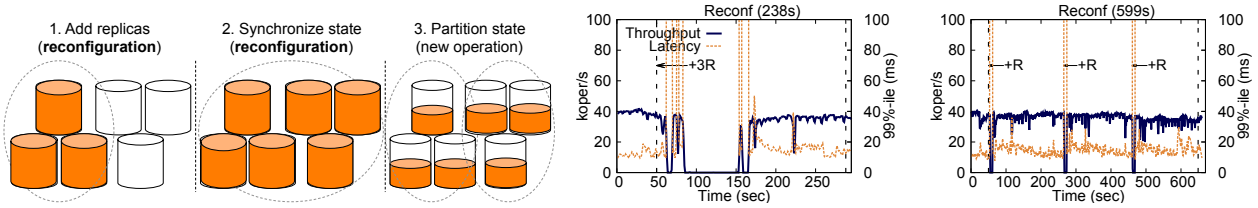
6.1.1 Partition transfer in existing RSMs

None of the classical SMR protocols support a primitive for dynamically creating partitions. However, this functionality can be added to an existing SMR-based service with minor or no modifications to the replication code. In the following we describe and evaluate two simple solutions that can be easily integrated in existing protocols. The objective is to characterize a baseline performance, which can be obtained with such straightforward approaches.

6.1.1.0.1 A client-based solution.

Our first candidate solution can be integrated to existing systems by adding three new operations on its interface, without modifying the replication code. As shown in Figure 6.2a, the idea is to have a special client (coordinator), that moves part of the state from a source RSM to a destination RSM, which will host the partition. To ensure there is no violation on the consistency of the service, the source RSM stops serving the transferred partition right after the first step, although this data is only deleted after it is installed in destination RSM.
Unfortunately, this straightforward design does not work well for large partitions. We implemented this protocol in a simple consistent key-value store built on top of BFT-SMaRt[1] [68] and conducted experiments to illustrate the mentioned problem. We used the YCSB benchmark read-heavy workload (95% reads and 5% writes) [116] to measure the performance of the system during a 4GB-partition transfer. Figures 6.2b and 6.2c show the throughput and the 99-percentile latency, calculated at every 2 seconds interval (see details about our experimental environment and methodology in §6.4), when the protocol described above is used for transferring 1024 blocks of 4MB and 256 blocks of 16MB.
The figures show that transferring a 4GB-partition takes 16.5 minutes, almost 21× more than a 4GB-transfer between two machines using rsync (see §6.4.1). More importantly, client operations accessing the block being transferred are blocked until the operation completes, causing spikes on the latency. This effect is more prominent when larger key-value blocks are transferred (Figure 6.2c), as

[1]Despite its name, BFT-SMaRt can be configured to tolerate only crash faults, using a protocol similar to Paxos [210]. This is the configuration used in this work.



(a) Reconfiguration-based partition transfer. (b) Adding 3 replicas at once. (c) Adding 1 replica at a time.

Figure 6.3: A reconfiguration-based protocol for partition transfer between RSMs with throughput and operation latency observed by clients during a 4GB-partition transfer operation.

the throughput decreases and latency spikes occur. When we transfer the whole 4GB-partition at once (not shown), the system stops serving client operations for 10 minutes.
In conclusion, despite the modularity of this naive protocol, it is clearly too disruptive and slow to be used in a practical elastic system.

A reconfiguration-based solution.

Another solution to dynamically manage partitions with no significant modifications on the replication code is to make use of SMR group reconfiguration, present in many practical protocols [210, 250, 241, 68, 284, 219]. This solution is being implemented for splitting groups in CockroachDB [7], an open-source version of Google Spanner [117] (which is described in §6.5).
SMR reconfiguration protocols allow the addition, removal and replacement of replicas within a single group. Adding a replica to a group means that the replica starts participating in the SMR ordering protocol, thus being able to process client requests. However, before processing such requests, new replicas must also be brought up to date by retrieving the state from the other members of the group. In contrast, removing a replica means that it stops participating in the ordering protocol.
Figure 6.3a illustrates the three main steps required for executing a split on a group of three replicas. First, three replicas are added to the source group using SMR reconfiguration. Second, and still within the reconfiguration, new replicas receive a copy of the RSM state. The experiments uses BFT-SMaRt "classical" state transfer protocol [68, 96], in which the new replica fetches the service state from one of the old configuration replicas and validates it using hashes obtained from other f replicas from the same configuration. Apart from the hash validation, this protocol is similar to what is employed in popular protocols and systems like Paxos [210], RAFT [250] and Zookeeper [181]. When the state transfer completes, a message is sent to all replicas to update their state metadata in a way that each half of the state is served by half of the replicas. In the end, each replica changes its replication layer to consider only the replicas responsible for the same part of the state as its group.
To understand the limitations of this protocol, we executed an experiment similar to the one described before, i.e., a 4GB-partition transfer. In this experiment, we were only concerned with the effect on the performance during the first two steps of the protocol, as the third one implies no substantial data transfer (see Figure 6.3a). Figures 6.3b and 6.3c show the effect on the throughput and latency during the reconfiguration of the replica group.
When three replicas are added in a single reconfiguration (represented by +3R in Figure 6.3b), the transfer of 4GB to the new replicas takes nearly 4 minutes, but the operation has huge negative effects on the system performance. In particular, the system stops processing requests for more than 60 seconds. This happens because as the source group size suddenly increases to six replicas, the required quorum (simple majority) for ordering requests increases to four replicas, of which one is necessarily a new one. Given that, a newly added replica is only able to process requests after it recovers the group state, the system will stop until the 4GB-state transfer is completed. These side effects of state transfer in RSMs were also studied in previous works [67].
A natural idea to avoid such undesirable effect is to add replicas one at a time, allowing a newly added replica to complete the reconfiguration procedure before adding another one. Figure 6.3c shows an

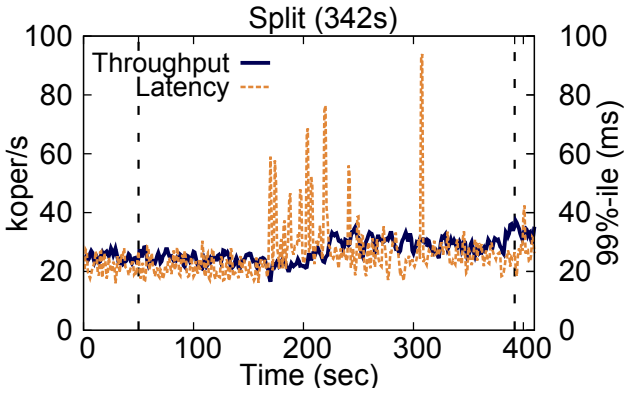


Figure 6.4: Throughput and operation latency observed by clients during a 8GB state redistribution in Cassandra.

execution of this setup, with each replica addition represented by a +R. The figure shows that, instead of having a long period in which the system stops, there are three short spikes/drops on the system latency/throughput. However, the drawback of this approach is that the full reconfiguration takes 10 minutes to complete (more than three minutes per reconfiguration), 2.4× more than with the previous strategy.
Overall, the key problem of the reconfiguration-based partition transfer is that the new replicas first have to receive the whole state, and only then split the group in two (discarding the unnecessary portion of the state).

6.1.2 Partition transfer in Non-SMR Databases

Given the impressive amount of work on elastic/cloud-enabled databases (see discussion in §6.5), an interesting question would be if the techniques employed in these services could be a solution for transferring partitions in RSMs. Systems like Cassandra [207], PNUTS [115] and Dynamo [130] support split and merge operations to handle changing workloads. However, even though the weakly-consistent replication protocols employed in these systems favor a scalable and elastic design, some recent works show that both their latency and throughput are significantly affected during reconfigurations of the replica set [116, 136, 121, 301].
We investigated how Cassandra [207] behaves when a three-replicas group hosting an 8GB database (in each replica) is scaled-out to six replicas (with 4GB each). The experiment considers the same read-heavy workload used before, with 200 clients accessing the database. After an initial warm up phase, we added three more servers, one at every 2 minutes, respecting the recommendations for the system [6]. To ensure consistency guarantees close to an SMR system, we configured the system to use a replication factor of 3 and quorums of 2 servers for both reads and writes.
Figure 6.4 shows the 99-percentile of the operation latency and the throughput during this experiment using disks. The results show that the whole process took nearly 6 minutes to complete. This happens because Cassandra employs a conservative design in which data transfers are judiciously done in background to minimize performance disruptions, which nonetheless occur, as can be seen by the latency spikes.
Other popular elastic databases implement such conservative design, and thus suffer from similar problems [116, 136]. In fact, the idea of performing state transfers in background have been used in other production systems for implementing replica recovery or reconfigurations (e.g., Zookeeper [181]), with the same rationale. In conclusion, despite these elastic databases being able to adapt their size to address changes in demand, such conservative design will always lead to a high setup cost for such systems.

6.2 Partition Transfer for RSMs

A key contribution of our work is the introduction of a partition transfer primitive and protocol in the SMR programming model. This primitive allows a replicated state machine G to transfer part of its state to another replicated state machine L, respecting the following requirements:

1. Protocol agnosticism: RSMs require protocols that order requests for implementing coordinated state updates and ensuring strong consistency. These protocols are complex to understand and far from trivial to implement [100]. A requirement of our solution is to not change the SMR protocols and use them as black-boxes for supporting ordered request dissemination in RSMs.

2. Preserve linearizability: A fundamental property of a RSMs is that it implements strong consistency. We want to maximize parallelism between operation execution and partition transfer without sacrificing the consistency of the service (i.e., linearizability [175]).

3. Performance: In the same way, we want to minimize the performance perturbations on the system during a partition transfer, while we minimize the time required for transferring a partition in modern datacenter setups.

In the following we first describe our assumptions about the environment and the elastic service (§6.2.1) and then present the partition transfer protocol (§6.2.2) and its correctness proof (§6.2.3). We conclude the section with a discussion of how to integrate our protocol with two existing approaches for executing multi-partition operations (§6.2.4).

6.2.1 System Model

Environment.

We consider a system with an unlimited number of processes, which can be either clients accessing a service or servers implementing the service, that can be subject to Byzantine faults. Therefore, the proposed protocol can tolerate state corruptions and thus be used in BFT systems [68, 96] as long as the total order multicast primitive employed tolerate Byzantine faults. Alternatively, if applied to a crash-tolerant replicated state machine, our protocol only tolerate crashes, but still preserving some state corruption validation due to its BFT design.[2]
Additionally, we require the standard assumptions for ensuring liveness of RSM protocols [210, 96, 250]. Processes communicate through fair channels that can drop messages for arbitrary periods but, as long as the message keeps being retransmitted, it will be received in the destination [225]. In terms of synchrony, we assume a partially synchronous distributed system model [141] in which the system can behave asynchronously for an unknown period of time, not respecting any time bound on communication or processing, but eventually will become synchronous.

Service.

Figure 6.5 illustrates the elastic SMR service. Clients invoke operations on a service by sending messages to the set of servers hosting it. Servers are organized in groups providing a stateful service as a replicated state machine. Each group of servers has n replicas and tolerates up to f simultaneous faults. We refer to group G as replicated state machine G. A RSM is accessed through a replica coordination protocol that ensures a strongly consistent operation. In practice, this means a consensus protocol will be used to ensure that requests are delivered to the replicas in total order (TO) [278]. We use TO-multicast and TO-receive to denote the transmission and reception, respectively, of messages in total order.

[2]In the evaluation of the protocol we considered its application to replicated systems tolerating only crash failures, since this allows reaching higher levels of system throughput and hence creates more stressful scenarios for the purpose of evaluating the reconfiguration performance. Therefore, in the evaluation we used BFT-SMaRt configured for crash fault tolerance.

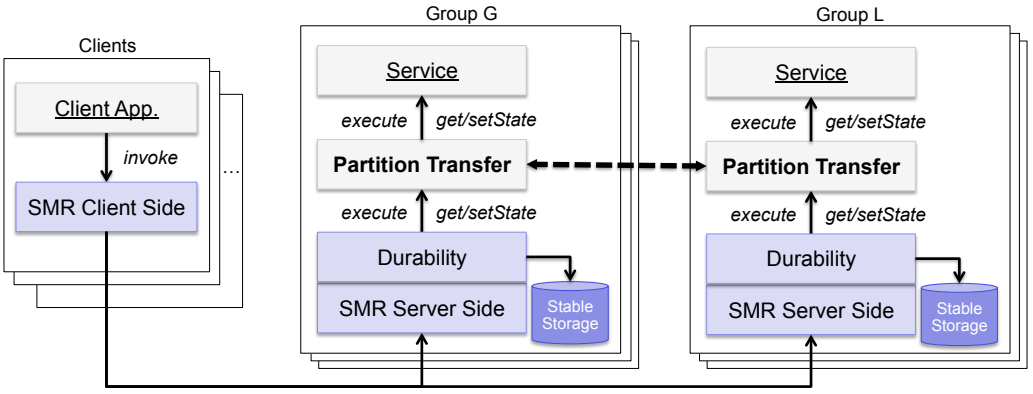


Figure 6.5: A partitionable and durable replicated state machine.

We assume that the service state S_G of group G can be partitioned in a number of closed partitions $s_1, ..., s_p$, in the sense that operations (and in particular, updates) submitted to the service affect only one of these partitions. For example, a key-value store can be partitioned in several key ranges such that each put/get will be executed on only one of them.
In the following we describe the partition transfer protocol considering only operations performed on a single partition. Later on we explain how the protocol can be integrated with multi-partition operations.

6.2.2 Partition Transfer Protocol

The partition transfer protocol is encapsulated in a primitive $ptransf(PS, L)$ that can be invoked in a replicated state machine G to make its replicas transfer the partition specified in PS (e.g., a key-range) to a replicated state machine L.
Figure 6.6 presents and illustrates the six-step protocol for implementing ptransf. The core idea of the protocol is to leverage the fact that all correct replicas (of both groups involved in the partition transfer) execute their operations in total order, mimicking a centralized server. In this way, it is possible to ensure that all correct replicas of G execute operations in PS until a certain point in the totally ordered history of executed operations, and then start redirecting further requests for this partition to the new group L (Step 4). Since partitions can be arbitrarily big, we use copy-on-write to make G store all executed updates in the partition specified in PS during its transfer to L (Step 2). Such updates are transferred after the state is received by a quorum of replicas in L (Step 5) and ptransf concludes (Step 6). The objective is to minimize interruptions to request serving in PS – these will occur only during the transfer of the updates Δ that were executed during the state transfer.
Several considerations can be made about the partition transfer protocol. First, our solution is completely modular with respect to the SMR protocol [210, 96, 68] or even the durability strategy [67] implemented in the system. This is quite important for reusing the already available protocols.
Second, we opted to perform full partition transfers between pairs of replicas, having one single replica in the source group transmitting to one single replica in the destination group. This design option makes our system bandwidth-efficient in multi-rack and virtualized environments. This pairwise transfer is executed after replicas of G connect with their corresponding paired replica in L (Step 2), which is done through a deterministic bijective function mapping between every replica from G to one replica from L. We envision scenarios in which the replicas of a group will be deployed in different racks (or physical machines) to avoid correlated faults and will have to transfer a partition to another set of replicas deployed in the same racks (or physical machines), ensuring such transfer will be done within the rack network boundaries, without using the (usually) oversubscribed network core.
Third, even if only crash faults are considered, we opted to provide a more general Byzantine-resilient protocol in which data corruptions are detected and recovered (through hash comparisons and state fetching, in Steps 3 and 5). This is important to ensure that even under the most uncommon failure

RSM Partition Transfer Protocol

(1) *ptransf(PS,L)* is TO-received by group G

(2) Each replica R_i of G sends the state S corresponding to partition spec. PS to its pair R'_i in L, and $H(S)$ to the other members of L

- From now on, every update on state S will be logged in Δ

(3) Replica R'_i in L accepts S when it is fully received together with f matching hashes from other replicas of G. R'_i sets its state to S and TO-multicast an ACK to group G

- If $f+1$ matching hashes (different from $H(S)$) are received, R'_i fetches a matching state from some replica from G or L

(4) When R_i of G TO-receives n-f ACKs from replicas in L, it sends Δ to its pair replica R'_i in L and $H(\Delta)$ to the other replicas of L

- From now on, replicas of G stop serving requests related to PS and redirect such requests to L
- Requests related to PS received by replicas of L are put on hold

(5) A replica R'_i in L accepts Δ only if it receives f matching hashes from other replicas of G. R'_i then applies Δ to its state and sends an ACK to the replicas of G

- If $f+1$ matching hashes (different from $H(\Delta)$) are received, R'_i fetches a matching update list from some replica from G or L
- The replicas of L start processing the PS-related requests that were on hold

(6) When a replica R_i of G receives n-f ACKs from L, it sends an ACK to the invoker of *ptransf(PS,L)*

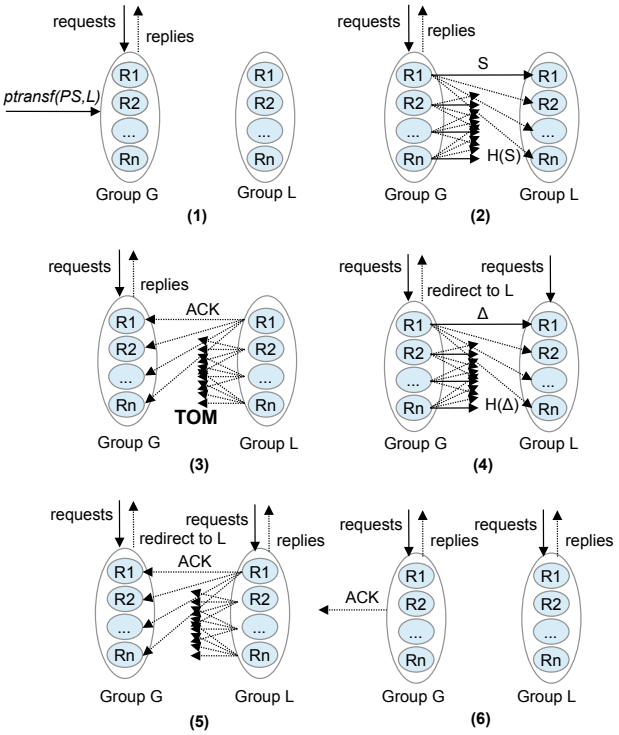


Figure 6.6: The partition transfer (*ptransf*) protocol.

modes [171] the destination group will still start accepting operations for the transferred partition with all its correct replicas in the same state. More specifically, at the end of the protocol at least $n - f$ replicas of L have the correct state for the partition. Note that the protocol does not preclude the case in which some correct replica of group L finishes the protocol with an invalid state. However, there will be a sufficient number of correct replicas in L with the correct state, and thus normal state transfer protocols for durable state machine replication can be applied, both in the crash [210, 250] and Byzantine fault models [96, 67]. This implies that all replicas in the group can start processing requests in the same state [278].

Fourth, consensus is employed only in two phases of the protocol (but encapsulated in the total order request delivery of the RSM): Step 1, when the ptransf primitive is invoked, and Step 3, when all replicas of L invoke RSM G to inform about the hash of the received state. Regarding Step 3, when the replicas of G receive $n - f$ matching ACKs from different replicas in total order, they send the same set of updates to the replicas in L. Notice that the $(n - f)$-th matching ACKs will be received in the same point of the execution history of all correct replicas of G, ensuring they will stop executing requests for PS in a coordinated way.

6.2.3 Correctness Argument

The ptransf operation must satisfy the following properties:

- **Safety 1**: When ptransf completes, the transferred partition will not be part of the source group state and will be part of the destination group state;
- **Safety 2**: Linearizability of the service is preserved by the partition transfer;
- **Liveness**: ptransf eventually completes.

Safety 1 is satisfied due to the fact that both the state (Step 2) and the updates (Step 4) are transferred. Moreover, after sending the updates (Step 4), the original group will not execute any other operation for the partition.

Safety 2 is a bit more tricky to show, since we need to take into consideration the definition of linearizability [175]. A service execution is linearizable if every execution history (containing requests and replies for client operations) is linearizable. We say that an operation is linearizable if the extension of a linearizable history with its request and reply still makes the history linearizable.
Linearizability is ensured in RSMs due to the fact that all requests are executed in total order by correct replicas. Consequently, every executed operation needs to produce a reply that considers all previously executed operations.
Let G be an initial group of replicas with state $S_{part} \cup S_{rem}$ and L a group of replicas that will receive a partition S_{part}. Recall that the system is partitionable, i.e., every operation either accesses S_{part} (the partition to be transferred) or S_{rem} (the remaining partition), as defined in §6.2.1. This means that the system is linearizable as long as the sub-histories containing operations for S_{part} and S_{rem} are linearizable [175].
Since the service provided by the RSM G ensures linearizability, the operations for S_{rem} are linearizable. Due to the same reason, the operations for S_{part} executed before $ptransf(part, \mathsf{L})$ are linearizable. When $ptransf(part, \mathsf{L})$ is received, the state S_{part} resulting from all previously executed operation for the partition is transferred to L (Step 2). Every operation for $part$ executed between Steps 2 and 3 is executed in G, and linearizability is maintained. These operations are stored in Δ and transferred to L in Step 4. After this point, G stops executing operations for $part$. As specified in Step 5, these operations are only executed by L after this step, when this group already received S_{part} and applied the updates in Δ. Consequently, after this point, these operations will reflect all previously executed operations for $part$, ensuring the linearizability of the partition.
Liveness is satisfied due to the fact that all six steps of the protocol terminate. This happens because (1) the total order multicast primitive employed in the protocol terminates in our system model (Steps 1 and 3); (2) the fair link assumption implies that both the state (Step 2) and the update list (Step 4) can be transferred in finite time; (3) if a replica does not receive the state or update list that matches f hashes, there will be $f + 1$ hashes matching some other state and list, and this state can be fetched (Steps 3 and 5); and (4) all correct replicas will send ACKs to the original replicas, and thus these replicas will receive $n - f$ messages to make progress (Steps 3 and 5).

6.2.4 Multi-partition Operations

In the following we describe how ptransf can be integrated with two existing approaches for supporting multi-partition operations, namely, the S-SMR protocol [258] and Spanner multi-partition transactions [117].

S-SMR.

S-SMR [258] considers a linearizable service with its state statically partitioned among several groups (partitions). Each group is composed by a set of replicas that implement a replicated state machine. Single-partition operations are executed only on individual groups, but the system also supports multi-partition operations, which require coordination of multiple groups.
More specifically, a multi-partition operation works as follows. First, the client determines which groups must participate in the operation and sends the operation to each of these groups using TO-multicast. Second, after ordering the requests (individually, on each group) the groups exchange all the necessary data to process the operation. Next, each group executes the operation and signals its completion to the other groups. After all groups involved were signaled, the result is returned to the client. The basic trick here is to block conflicting operations and execute them in sequence, preserving linearizability [258].
To integrate ptransf with S-SMR, we need to make two modifications on the original protocol. First, after a reconfiguration, the system needs to update the partition info to allow clients to find the new partitions. Second, operations can not be processed while Steps 4 and 5 of ptransf are being performed in the source group G. Operation execution can only be resumed after this group receives

$n - f$ totally-ordered ACKs from the destination group L. This guarantees that the partition state and its differences were received by L, ensuring the consistency of the service.
When the partition transfer finishes, source group G may still hold multi-partition operations that can not be executed since it is no longer responsible for the data required by them. In this case, G signals the other groups involved in the operation, and the client receives a redirect message, restarting the multi-partition operation in the updated groups.

Spanner.

Similarly to S-SMR, in Spanner [117] the application state is partitioned in tablets hosted on several groups. Each group is composed by a set of replicas that implement Paxos state machines to maintain shards of a database.
The steps to process a multi-partition transaction are the following. First, a client communicates with a proxy location to determine which groups maintain the data touched by the transaction. Second, the client retrieves this data from the groups, acquiring locks for them. Next, the client executes its transaction locally, chooses one of the groups involved in the transaction as a coordinator group C, and sends the result and the id of C to all groups involved in the transaction. Finally, group C coordinates a two-phase commit with the other groups for committing the transaction.
To integrate Spanner transactions with ptransf we need to slightly modify Steps 4 and 5 of our protocol to account for the way locks are managed in Spanner. More specifically, while the partition is being transferred, its associated data must be locked. The source group only releases the locks after receiving $n - f$ totally-ordered ACKs from the destination group. This guarantees the consistency of the service as the partition state and its differences are received by the other group. When a client tries to access the data in the transferred partition, it receives a redirect message and aborts the multi-partition transaction. Eventually, the proxy location server is updated and the client can re-execute the transaction in the updated groups.

6.3 Implementation

We implemented our partition transfer protocol in an elastic storage infrastructure called CREST, whose architecture is depicted in Figure 6.7.

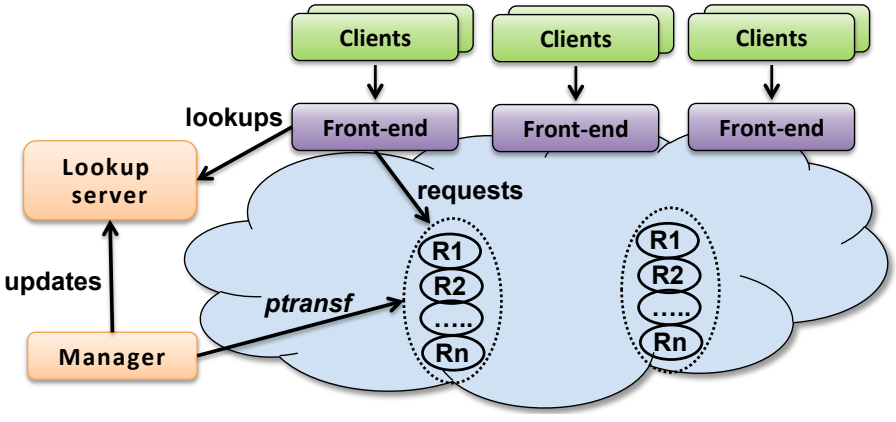


Figure 6.7: CREST architecture.

In CREST, clients send requests to stateless Front-ends exposing a key-value interface including operations such as GET and PUT. The Front-ends use the Lookup server to discover which group serves a certain key. All data is sharded across the groups. The Manager triggers reconfigurations by sending ptransf commands to the groups.
We implemented the ptransf protocol on top of the BFT-SMaRt [68] replication library. Our implementation follows the model described in Figure 6.5, with the partition transfer layer implemented

between the durability management [67] and the service (in this case, a key-value store). The pairwise transfer of the partition state and update list (see Steps 2 and 4 in Figure 6.6) is implemented over dedicated sockets, to minimize interference with the normal processing of client operations. To serialize the key-value store partitions we used a serialization library called Kryo [13]. All updates and partition transfers are written to the replicas log, in durable storage. We parallelize the writing of state to the storage device (including buffer flushing) with other steps of the protocol, leveraging the fact that (1) the original group will only delete the transferred state after Step 5 and (2) the receiving group disk is idle during a state split since it only starts executing and logging requests after Step 5. The Lookup server and the Front-ends were implemented as Java servers, and the Manager as a Java client. As already mentioned, a Front-end is just a soft-state proxy. The Lookup server holds a map between intervals of keys and groups. For each Front-end request, the Lookup server returns a key interval and a group. In this way, the number of lookup operations can be minimized as the Front-end caches key locations. The Manager is responsible to send ptransf commands to the groups in order to perform reconfigurations. After every reconfiguration process, the Manager updates the Lookup server map.

6.4 Evaluation

We evaluate our partition transfer algorithm within CREST by assessing the duration of a partition transfer and how it affects the service throughput and operation latency observed by clients generating different workloads on the system.

Setup.

All experiments were conducted in a cluster of 18 machines interconnected by a gigabit ethernet switch. Each machine has two quad-core 2.27 GHz Intel Xeon E5520 with 32 GB of RAM memory, a 146GB 15k-RPM SCSI disk and a 120GB SATA Flash SSD. The machines run Linux Ubuntu Trusty with kernel version 3.13.0-32-generic for x86_64 architectures with Oracle Java 1.7.0_80-b15. In these experiments we avoided using VMs and dynamic resource configuration managers to capture the performance of the partition transfer protocol without the VM setup overhead.
Unless stated otherwise, the experiments were executed with BFT-SMaRt configured for crash fault tolerance, with groups of three replicas (tolerating a single fault).

Methodology.

Our setup considers an 8GB database where small-string keys were associated with 4kB values. We used two YCSB workloads in our experiments [116]: the read-heavy workload, with 95% of get and 5% of put operations (95/5), since it is similar to what is reported in several production systems [79, 115, 117] and in some related works (e.g., [301]); and the write-heavy workload, with an equal distribution of puts and gets (50/50) to exercise our protocol under a heavy update load. Clients access keys following a Zipfian distribution. This setup and workload creates a reasonable but demanding scenario for an SMR-based storage system [68, 67, 70].
Our experiments consider partition transfers of 4GB (half of the KV-store), either in a single ptransf execution or in multiple executions of ptransf, for transferring smaller blocks of 16MB and 256MB (approximately 256 and 16 protocol executions, respectively).

6.4.1 Partition Transfer on an Idle System

Our first experiment measures the time our protocol takes to transfer 4GB of state between two RSMs without any client-imposed workload. The results are presented in Table 6.1, which also contains the duration of the same transfer using the client- and reconfiguration-based solutions described in §6.1.1, considering the state maintained in both SSDs and disks. We also present, as a reference, the duration

of a 4GB-file transfer between two machines using rsync [23], a widely-used tool for synchronizing files between two machines.

System	Disks	SSDs
ptransf	54 ± 1	64.40 ± 2.97
client (4MB)	802.3 ± 3	823.3 ± 1.1
client (16MB)	855 ± 3.78	873 ± 5.0
reconfig (+3R)	201 ± 8.1	209.6 ± 11.5
reconfig (+R+R+R)	294 ± 4	307.3 ± 2.3
rsync	44.80 ± 1.30	52.20 ± 0.84

Table 6.1: Duration of a 4GB partition transfer (in seconds) using ptransf and alternative solutions (see §6.1.1) in an idle system.

The results show that ptransf is 18× and 4×-9× faster than the client and the reconfiguration-based solutions, respectively. On the other hand, ptransf is 23% and 20% slower (for SSD and disk, respectively) than a two-machine synchronization using rsync.
Notice that the reported performance using disks was better than using SSDs. This happens because our disks have a better throughput than our SSDs for sequential writes (130 vs. 120 MB/s).

6.4.2 Partition Transfer on a Saturated System

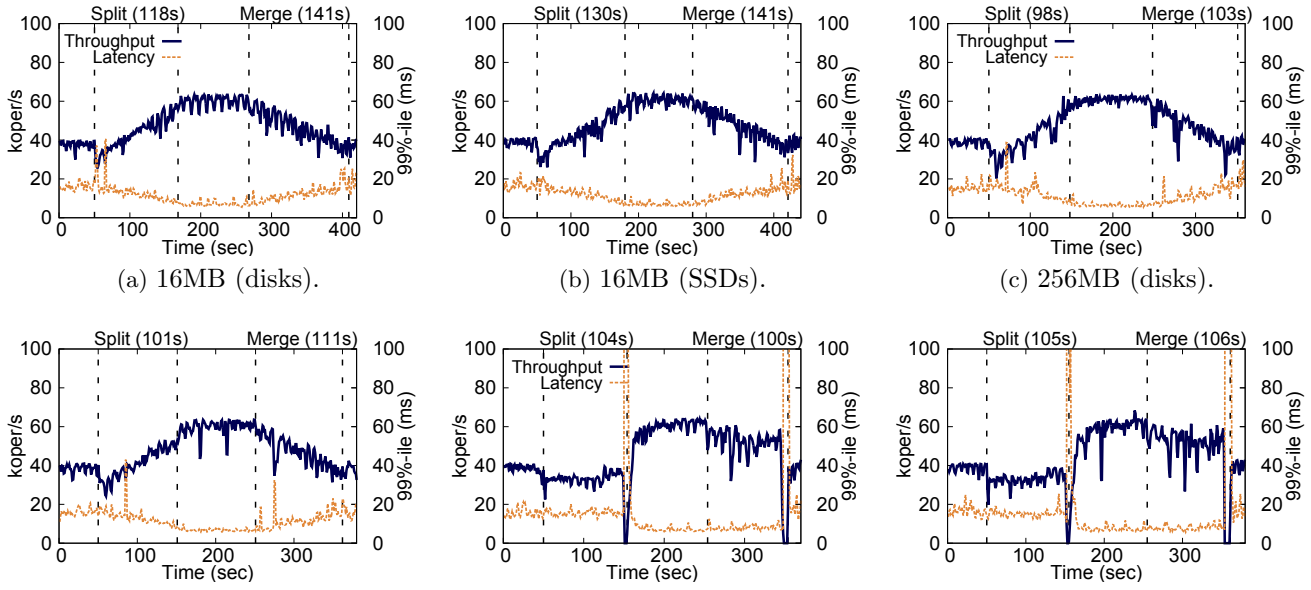
The next set of experiments aims to shed light on the impact that a partition transfer can have on the performance of a saturated system. The objective is to understand how triggering a partition transfer (e.g., for scaling-out) under critical conditions affects the latency and throughput of the system.
In order to define the conditions for system saturation, we progressively launch a number of clients until the system reaches its peak throughput for a single replica group. Adding more clients after this point only increases the latency. We identified that the system achieves the peak throughput for read-heavy and write-heavy workloads with 140 (≈40000 4kB-oper/s) and 70 (≈9000 4kB-oper/s) clients, respectively. The observed peak throughputs are similar with disks and SSDs.
In all experiments, when the system reaches its peak throughput we start a partition transfer to split the state of the system to another group. One hundred seconds after the split completes, we invoke another partition transfer to merge the state of the second group back to the first.

Read-heavy workload.

Figure 6.8 shows the throughput and the 99-percentile operation latency (obtained from 2-second intervals) observed by clients running the 95/5 workload using disks and SSDs.
Three aspects are worthy of consideration from these executions. First, the state split tends to be faster than the state merge (the second transfer) using both disks and SSDs. This happens because the split transfers the partition to an idle group (not yet receiving client operations), while the merge transfers the state to a busy group. Second, as in idle setups, the partition transfers are faster with disks. Third, there are less spikes in the throughput and in the operation latency when the partition is transferred in small blocks. For instance, with a 16MB block size, the throughput starts to increase few seconds after the split initiates, as clients start sending commands to the second group, decreasing the load on the first. In contrast, with a 4GB block size (a single ptransf execution), the throughput starts to increase only a few seconds after the split completes, while the operation latency increases significantly during the partition transfer. This happens because the group is fully loaded with client operations while transferring the whole partition and keeping track of the updates being executed.

Write-heavy workload.

Figure 6.9 shows similar executions but now considering the 50/50 workload.



(a) 16MB (disks). (b) 16MB (SSDs). (c) 256MB (disks).

(d) 256MB (SSDs). (e) 4GB (disks). (f) 4GB (SSDs).

Figure 6.8: CREST throughput and operation latency in saturated conditions with reconfigurations using disks and SSDs. Read-heavy (95/5) workload.

The behavior observed in these experiments is similar to the discussed for the read-heavy workload, with two noticeable differences. First, the latency is slightly higher and the throughput is substantially lower than in a read-heavy workload. Second, there are more latency spikes during splits and merges, both for disks and SSDs. This happens because now there are more updates, which imply more operations being written to write-ahead log of the SMR [67] and increased I/O contention with the partition write at the receiving group. These spikes tend to be more common during merges than during splits. The explanation comes from the different nature of splits and merges. During a split, the state is transferred to an initially empty RSM, which is progressively being loaded, as blocks of the partition are transferred. During a merge, on the other hand, the state is transferred to a group already loaded, which causes higher I/O contention. This should not be a problem in practice since splits are executed under high loads, while merges are triggered for consolidating resources, when the system is mostly idle.

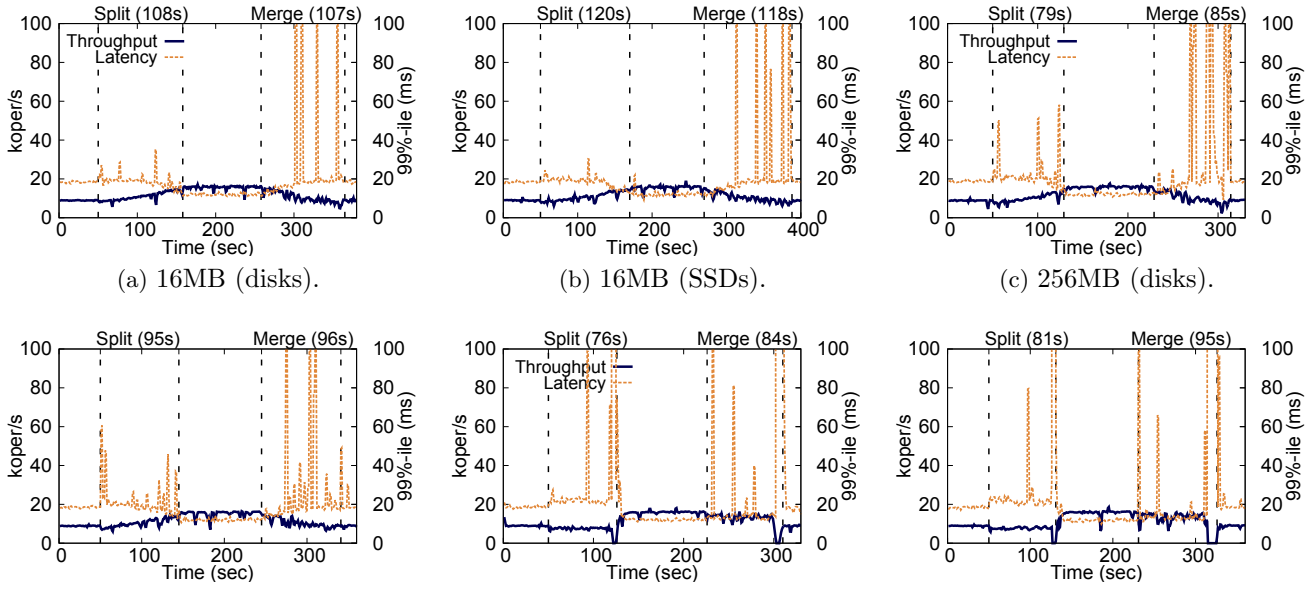
Consolidated results.

We now show consolidated results for 10 executions of splits and merges using each configuration, comprising almost 4 hours of partition transfers. As in previous experiments, these results (presented in Figure 6.10) consider different workloads and block sizes. However, from here on we only show results for setups using disks due to space constraints and because the results using SSDs lead to the same insights.

Figure 6.10a reports the 90th and 99th latency percentiles for periods with and without partition transfers (noted in figures as “Split” and “Merge” operations for different block sizes and “No PT”, respectively) for 95/5 and 50/50 workloads. The 90%-iles latency represents the behavior that most clients observe during a split or a merge, while 99%-iles capture the effect of spikes on the service level metric.

The results confirm the trends observed in the previous section. In particular, splits and merges have no significant effect (when compared to the “No PT” situation) on the 90%-iles latencies under the 95/5 workload. This is not true for write-heavy workloads, as the latency of merges are consistently higher (due to I/O contention).

As expected, the 99%-iles latencies are affected during splits and merges for all block sizes and work-



(a) 16MB (disks). (b) 16MB (SSDs). (c) 256MB (disks).

(d) 256MB (SSDs). (e) 4GB (disks). (f) 4GB (SSDs).

Figure 6.9: CREST throughput and operation latency in saturated conditions with reconfigurations using disks and SSDs. Write-heavy (50/50) workload.

loads. However, the values during splits are still under 60 ms, which is way bellow real-world SLAs defining the 99%-iles under 100 ms [115, 130]. The effect is much more dramatic for merges, for the reasons discussed before. However, such effects have few practical implications as merges are usually executed in idle systems (as also discussed before).

The only exception to the general trends discussed before is observed when using 4GB blocks. In this case, the 99-%-ile latency increases 100× or even more, as clients block during the transference of the update log in Step 4 of ptransf. This is illustrated in Figures 6.8e (gaps at seconds 150 and 350) and 6.9e (gaps at seconds 120 and 290).

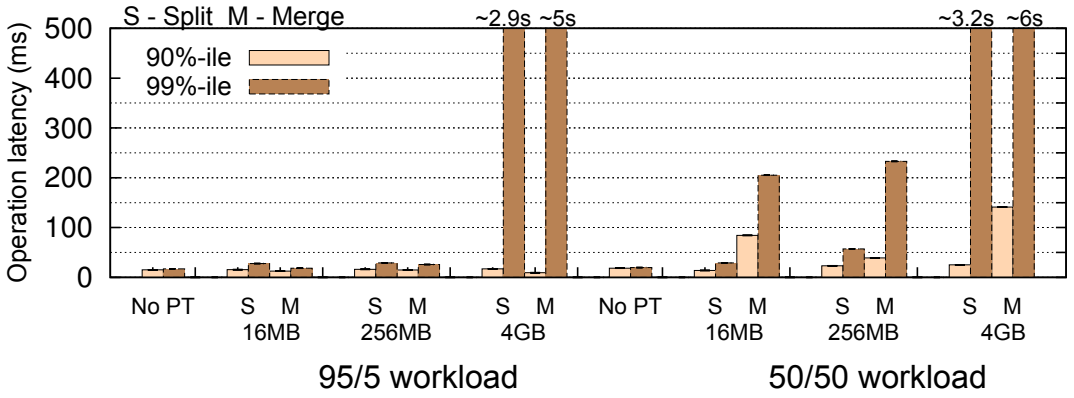
Figure 6.10b presents the average duration of split and merge operations (together with standard deviation) during the previous experiments. The results show that when considering a specific configuration, the amount of time required for executing the partition transfer is stable across different executions, as attested by the negligible standard deviation. Additionally, the results show that transferring 16MB-blocks lead to higher partition transfer durations than when using other block sizes in all configurations. This happens mostly because the ptransf protocol runs more times, leading to more protocol messages being exchanged and more synchronization points (Steps 4 and 6 in Figure 6.6). However, the duration of splits and merges are similar for blocks of 256MB or 4GB (whole partition in a single transfer). This indicates that increasing the block size after a certain value does not lead to faster partition transfers, instead, it only makes the latency worse (as shown in Figure 6.10a).

Comparison with alternative solutions.

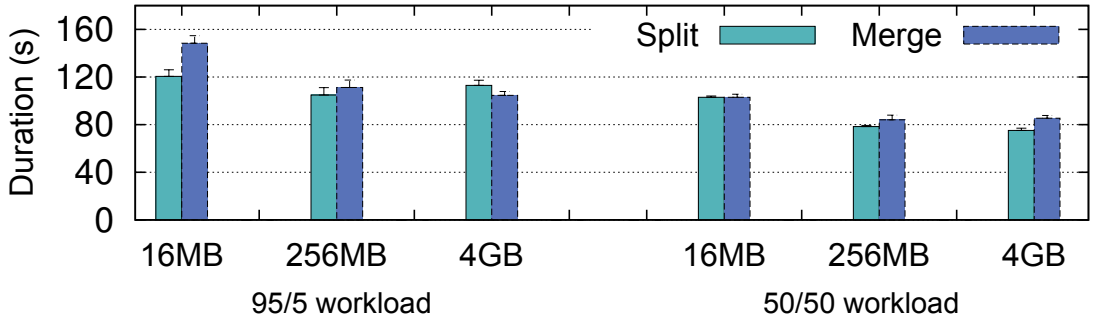
Table 6.2 compares the duration of a split using the best configuration of ptransf (256MB blocks) and the alternative solutions discussed in §6.1.1 for a saturated system. By comparing this table with the disks column in Table 6.1 it is possible to see that all protocols have a noticeable increase on the duration of split, with slower protocols (client-based) being less affected than faster ones (ptransf).

The results show that a split using ptransf is 9.5× and 11.7× faster than the client-based solution (Figure 6.2), and 2.3× and 5.4× faster than the reconfiguration-based solution (see Figure 6.3) for a read-heavy workload. The difference is even bigger with a write-heavy workload.

Even more importantly than these results is the fact that our protocol causes less perturbations on the latency and throughput observed by clients, as can be seen by comparing the executions of Figures 6.2



(a) Operation latency during partition transfers (groups of three replicas).



(b) Duration of partition transfers (groups of three replicas).

Figure 6.10: Latency during 4GB-partition transfers and the duration of such transfers using disks and groups of three replicas ($f = 1$) and considering different block sizes and workloads.

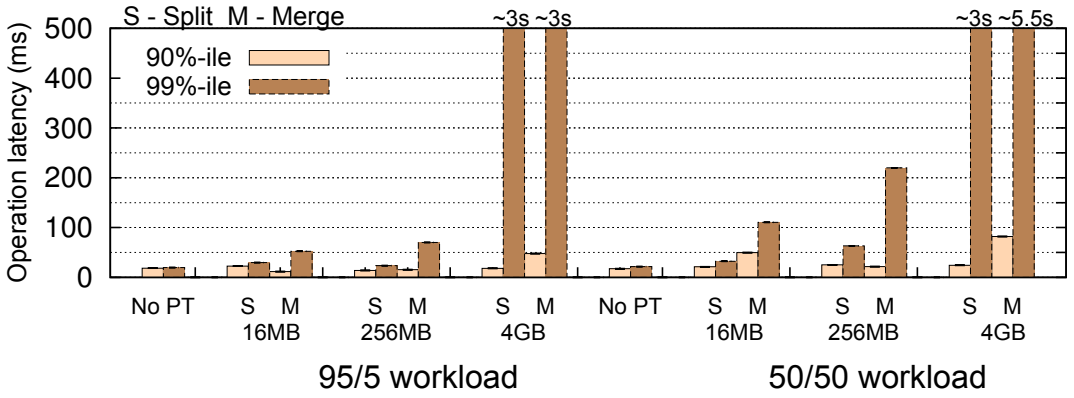
System	read-heavy	write-heavy
ptransf	104.8 ± 6.3	78.4 ± 0.9
client (4MB)	999 ± 15.9	871.3 ± 11.7
client (16MB)	1236 ± 16	1198 ± 9
reconfig (+3R)	243.5 ± 6.6	270.4 ± 2.3
reconfig (+R+R+R)	575 ± 21	623 ± 10.4

Table 6.2: Duration of a 4GB partition transfer (in seconds) using ptransf and alternative solutions (see §6.1.1) in a saturated system using disks for read- and write-heavy workloads.

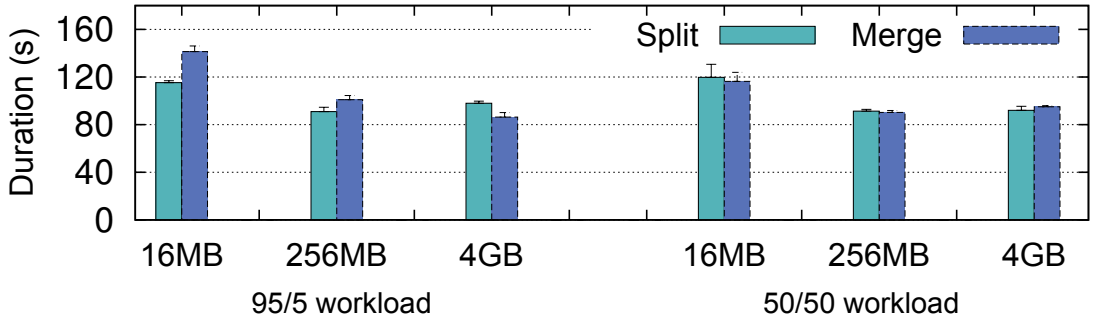
and 6.3 with splits presented in Figures 6.8 and 6.9. The only exception is in Figure 6.3c, for the three single-replica reconfigurations, in which the effects are negligible, but the transfer latency is 5× higher than with our protocol.

6.4.3 Partition Transfer in Bigger Groups

The results presented up to this point consider groups with three replicas, that tolerate a single server failure ($f = 1$). Figure 6.11 presents results for similar experiments, but considering groups of five replicas ($f = 2$), to observe the behavior of our protocol when considering bigger groups.
When compared to the results for groups of 3 replicas (Figure 6.10), the results in Figure 6.11 show exactly the same trends, despite some small variations in the latency.
Regarding the duration of the partition transfers, the results are also similar to the three replica setups.



(a) Operation latency during partition transfers (groups of five replicas).



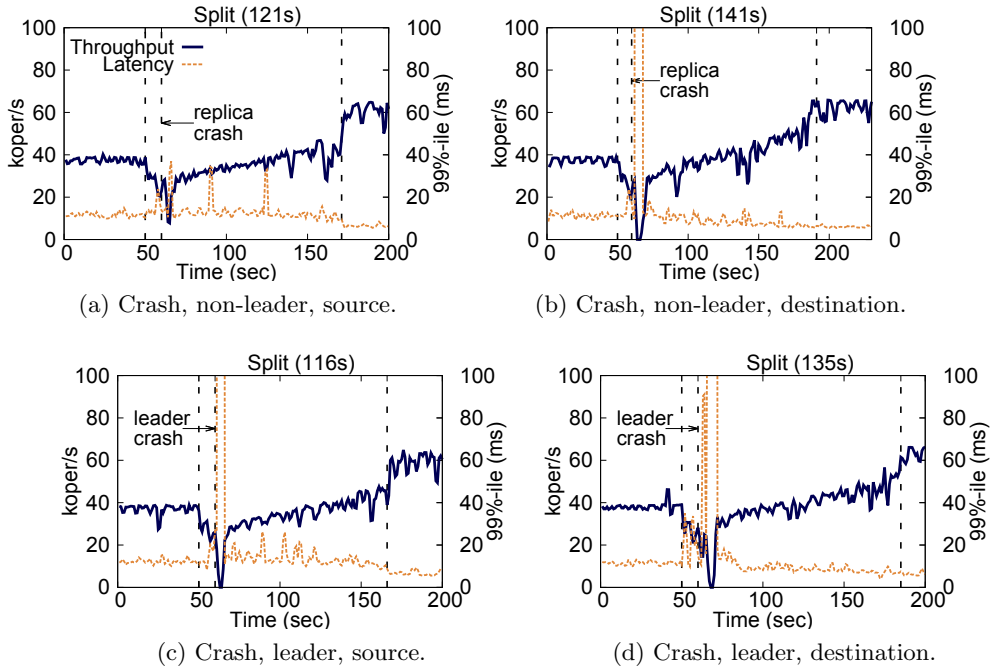
(b) Duration of partition transfers (groups of five replicas).

Figure 6.11: Latency during 4GB-partition transfers and the duration of such transfers using disks and groups of five replicas ($f = 2$) and considering different block sizes and workloads.

This happens because our protocol transfers state of replicas in parallel, pairing each replica on the source group with another replica on the receiving group. This design makes the data transfer phases of the protocol (Steps 2 and 4 in Figure 6.6) – which dominate the duration of a transfer – mostly independent from the group size.

6.4.4 Faults during the Partition Transfer

In this section we discuss how replica failure scenarios affect the ptransf protocol. We consider four failure scenarios: (a) a non-leader replica of the source group crashes, (b) a non-leader replica of the destination group crashes, (c) the leader replica of the source group crashes, and (d) the leader replica of the destination group crashes.
In this experiment, we considered a group of three replicas, a read-heavy (95/5) workload and blocks of 256MB. The experiment focuses on the split operation since it is normally executed when the system is under stress and needs to scale-out as fast as possible. In all failure scenarios we crash the target replica 10 seconds after the split begin and use a timeout of 2 seconds for suspecting a leader and starting a view change. Figure 6.12 presents representative executions for the four scenarios.
When comparing the executions with a non-faulty execution (left half of Figure 6.8c), it is clear that failures have a noticeable effect on the partition transfer.
Comparing the scenarios where a non-leader replica crashes (Figure 6.12a and 6.12b), the negative effects on the system performance are more prominent when the replica belongs to the destination group. This happens due to a combination of two factors. First, during a split the operations targeting the partition being transferred are progressively being redirected to the destination group, with each



(a) Crash, non-leader, source.

(b) Crash, non-leader, destination.

(c) Crash, leader, source.

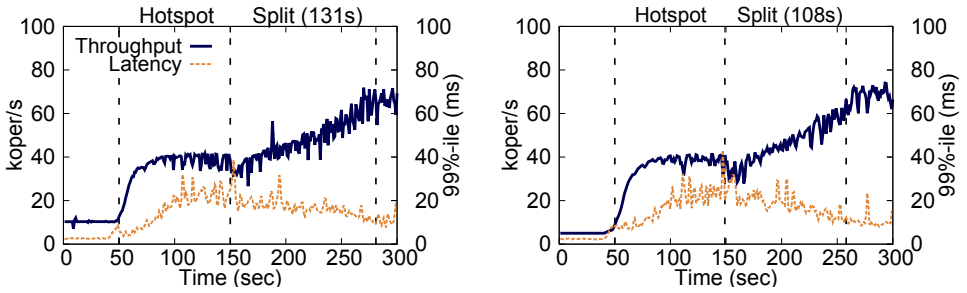
(d) Crash, leader, destination.

Figure 6.12: Four failure scenarios during a split using a 256MB block size and disks.

block transferred. Second, a high percentage of the reads performed by clients on the two surviving replicas of the destination group are not completed using the consensus-free optimized read, and need to be retried using the total-order protocol [68, 96].
Crashing the leader replica of both source and destination groups (Figures 6.12c and 6.12d) leads to the same effect in the system: the group with the faulty replica stops processing requests for 2 seconds (the timeout value), until a new leader is elected and normal processing is resumed.
In summary, the results show that the performance of the partition transfer protocol is robust against failures, and the most visible negative effects are due to performance degradation caused to the ordering protocol of the RSMs.

6.4.5 Partition Transfer in a Hotspot

In previous experiments we performed splits and merges on saturated RSMs to evaluate how our protocol works in a saturated system. In this final experiment we discuss the effects of doing a partition transfer on a hotspot group, i.e., a group of replicas that experiences a sudden and large increase in its workload. We create such hotspot by uniformly increasing the number of clients, from 20 to 200, over a hundred seconds interval, which also results in a tenfold latency increase (from 2.5 ms to 25 ms). This hotspot scenario is similar to the one used in [301], and was inspired on the statistics of the "9/11 spike" experienced by CNN.com [215], where the workload increased by an order of magnitude in 15 minutes. For this experiment, we consider groups of three replicas and a read-heavy workload.
Figure 6.13 shows the throughput and the latency when a split is performed on the hotspot group using two block sizes. As can be observed, the split causes a similar effect on the throughput and the latency when the group is saturated (Figure 6.8c and Figure 6.8e) or heavily saturated (hotstop). Using 16MB blocks to transfer the partition makes the split 20% slower than using 256MB blocks, but the overall effect on latency and throughput is the same.
In conclusion, the results show that our ptransf implementation allows a 8GB storage system to double its capacity when subject to unusual high demand without any significant performance disruption, taking less than two minutes to scale-out.



(a) 16MB block size.

(b) 256MB block size.

Figure 6.13: Scale-out in a hotspot group using disks.

6.5 Related work

SMR scalability

Several SMR protocols support the addition, removal and replacement of replicas at runtime [210, 250, 241, 68, 284, 219]. However, these reconfigurations only change the set of replicas in a single group, and do not improve the performance of the system since the protocols used for ordering requests are inherently non-scalable. To the best of our knowledge, we are the first to propose a well-defined primitive and protocol for sharding RSMs at runtime.

Different techniques have been proposed to improve the scalability of SMR-based services. Some works try to remove bottlenecks both from the ordering protocol [229, 241], its implementation [276, 58] and the execution of requests [198, 169, 192]. For example, protocols like Mencius [229] and Egalitarian Paxos [241] try to spread the additional load of the primary replica among all the system replicas. In a complementary way, Santos and Schiper [276] and Behl et al. [58] shows that it is possible to substantially improve the performance of an SMR implementation by architecting these systems using multi-threaded architectures. Similarly, several works try to parallelize the execution of requests for taking advantage of multi-core servers and improve the RSM performance. The techniques range from identifying independent RSM commands that can run in parallel without endangering determinism [198], to executing the requests (using multiple threads) before ordering them [169, 192]. Ultimately, these approaches help addressing the request ordering and execution bottlenecks (and thus can be combined with CREST to improve the performance of a single group), but the fundamental limitation of every replica executing every operation still remains.

A third group of works aims to scale SMR-based services by dividing its state among several partitions, implemented by (mostly) independent RSMs [258, 253, 163, 117]. Bezerra et al. [258] propose a technique for executing atomic operations spanning multiple partitions still ensuring linearizability. In [253] it is proposed a storage architecture that follows a partition approach and tolerates Byzantine failures. This architecture enables transactions across partitions. Although both [258] and [253] use partitioning to address the scalability of SMR-based services, there is no support for creating such partitions dynamically.

Elasticity in SMR-based systems.

Scatter [163] is a consistent distributed key-value store where key ranges are served by groups of replicas. Split and merge reconfigurations are available, but the paper does not mention how the state transfer between partitions is realized, neither measures the impact of such data transfers (the reported values suggest that only a trivially-small state is used in the experiments). Additionally, the Scatter partitioning algorithm works only for adjacent groups in its Chord-like architecture, requiring substantial modifications to the Paxos protocol to implement a multi-group 2PC commit. Our solution, on the other hand, aims for fast and predictable elasticity with multi-gigabyte partitions, targeting

thus a general limitation of SMR systems. Moreover, it can be implemented on top of any SMR protocol.
Spanner [117] is a globally-distributed database that shards data across many sets of Paxos-based state machines in Google datacenters. Although not detailed in the paper, Spanner allows shards to be transferred in order to balance load or in response to failures. During these transfers, clients can still execute transactions (including updates) on the database. Similarly to most elastic database systems [207, 115, 130], Spanner transfers partition data slowly to minimize the impact of such reconfigurations on the system performance. When the remaining data to be transferred is sufficiently small, the system uses a multi-partition transaction to move the metadata of the shard between the two groups. Although the protocol appears to protect the safety of the database, the liveness is not guaranteed since moving data in Spanner may take an unbounded amount of time if the update rate is higher than the transfer rate. In this chapter we proposed a specialized abstraction and a (safe and live) protocol for transferring partitions as efficiently as possible, and explained how it could be integrated with the multi-partition transactions of Spanner.

Elastic databases.

There is a large stream of works from the database community for implementing elasticity in existing systems [282, 123, 124, 296, 144]. These works propose solutions to split and merge state of sharded databases without violating the ACID properties, and can be broadly divided in terms of the database architecture they consider. Some systems employ shared storage architectures [124, 123], where the database nodes persist data on some shared "always available" infrastructure. In such architectures, when a server is added there is no need to perform a partition transfer, being sufficient to copy only the database cache and active transactions. Other works focus on shared nothing architectures [144, 282, 296], where each partition is kept on different nodes. In this case, reconfigurations require a partition transfer. However, existing works only consider transferences between two servers, not between two groups of servers, as is required for RSMs.

6.6 Conclusion

This chapter described a partition transfer protocol for implementing elasticity in replicated state machines. Our protocol minimizes (1) the time required to transfer the partition between two replica groups and (2) the negative effects of this data transfer on the operation latency observed by clients. The proposed protocol can be integrated in any SMR consensus algorithm (e.g., Paxos [210], PBFT [96] or RAFT [250]), since it operates on top of the ordering protocols. Furthermore, it allows the dynamic creation of partitions in different replica groups, complementing some recent works on scalable state machine replication [258, 163, 253].
The proposed protocol was implemented and evaluated in a key-value store. In our experiments, a prototype system using our protocol was able to double its capacity with minimal performance degradation, showing that it is possible to have elastic reconfigurations even for non-trivial partition sizes (e.g., 4GB).

Part II

Resilient distributed storage

Chapter 7 Janus – A User-Defined Cloud Storage Platform

7.1 Introduction

The high volume of data that has been generated in the last years has increased the demand for the use of cloud storage services. This happens because such services allow users and organizations to store their data following a pay-as-you-go cost model at the same time that it frees them from infrastructure-related concerns. However, one has to deal not only with the amount of data to store, but also with the diversity of it, especially companies.
Different kinds of data have different storage requirements, for which companies are willing to spend different quantities of resources (e.g., money, bandwidth). The data of the company's clients, such as their address, account password or credit card number, is clearly more critical than the photos of the company's Christmas dinner.
Despite the interest of the company in storing both of these kinds of data, we advocate that it would be willing to pay more to ensure the privacy, security and durability of the clients' data than of the photos. In this example, the company has only two types of data, and the only differentiating requirement used is their storage reliability. Nonetheless, there are types of data that have other requirements, such as latency-related SLAs, the data locality restrictions or even the need to support multiple concurrent writers.
Janus is a platform that finds the best way to store data in the clouds according to given user-defined requirements. It maintains informations of several storage services from the commercial cloud storage providers (e.g. read/write/storage prices, latency, etc.), and it is uses diverse storage techniques to provide different guarantees. To be able to find the best storage solution, Janus permits users to specify their workload and/or requirements, and according with them, finds the most appropriate storage techniques and cloud services to use.
The system can store data in two back-ends (*single-cloud* and *multi-cloud*), and it is able to tolerate both *Byzantine* and *crash* faults in the clouds. It is also storage-efficient, through the use of *erasure-codes*, and ensures the privacy of the data by encrypting it at client-side. We designed the platform to allow the storage of the data in the clouds without requiring the intervention of no dedicated server (i.e., the system is *serverless*), meaning that all data manipulation is executed at the client site, not depending on the Janus server. This design also permits the client identity to be unknown by the clouds, since the cloud provider accounts are created and managed by our system, without any client-specific information.
There are some solutions that store data in the clouds using different storage techniques. Synchronization services like DropBox [139] use a single cloud to store the data and do not allow clients to choose what that cloud is. There are some works that allow users to statically define the clouds that will be used [30, 54, 65, 66, 134, 172, 297], and some that also allow the definition of some system parameters (e.g., number of faulty cloud tolerated, cache size) that influence the behavior of the system to meet some goal [111, 233, 264]. Nonetheless, Janus takes the user-defined storage concept [94] further by permitting clients to specify their high-level requirements, and making the storage techniques and cloud services configuration transparent to the client.
Given this, the main distinguishing features of Janus are:

- a *solver* capable of finding the most appropriate cloud storage solutions by using the cloud services information it maintains and the given storage requirements.
- a *serverless platform architecture design* that permits the client to be anonymous to the clouds.

7.2 Janus Overview

JANUS is a platform that permits the creation of virtual disks (called volumes) to store data in cloud storage services that are compliant with a set of specifications defined by the users, i.e., a set of requirements and constraints. The architecture of the platform is designed to make all the process efficient in terms of management effort, storage costs, and performance, while ensuring also the compliance with the user requirements. Figure 7.1 presents the architecture of the platform.

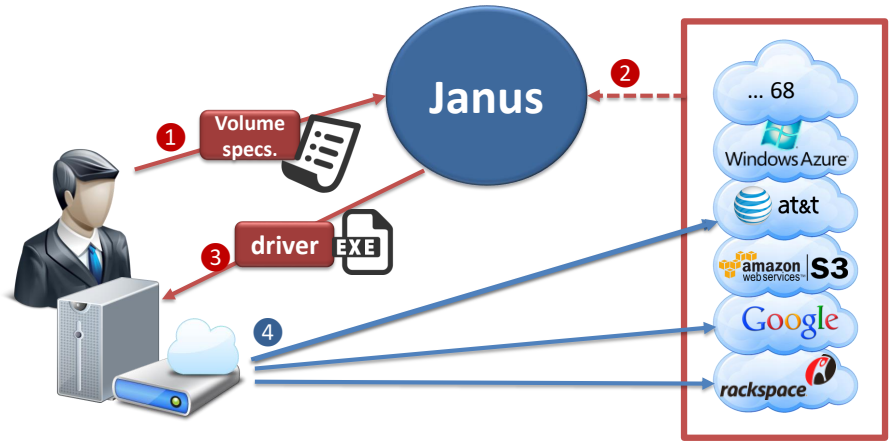


Figure 7.1: JANUS architecture.

The JANUS platform is essentially composed of two components:

- *janus server* is responsible for finding the most adequate storage configurations given the volume specifications. This server is also responsible for the creation and management of the cloud accounts, and for maintaining informations about possible clouds in use (i.e., available locations, latency, prices, etc.).
- *virtual disk driver* accommodates all the different storage profiles of the client and is responsible for managing the system's data-plane. It is installed in the client's local machine and directly interacts with the clouds used for each profile.

As shown in Figure 7.1, to create a virtual disk with certain storage requirements, a user contacts the JANUS server for giving its requirements and constraints (1). The server finds the best possible storage profiles for that request and returns them to the user (2), which picks one of them. A driver with the chosen profile is then downloaded and installed in the client's local machine (3). After that, all the data present in the JANUS virtual disk is stored in the clouds according to the user-defined requirements (4).
This platform design allows the system to have some interesting capabilities in terms of privacy, performance and fault tolerance. The fact the virtual disk driver is serverless, i.e., it interacts directly with the clouds without contacting the JANUS server for any operation, allows the system data-plane to operate directly with the cloud services interface without requiring any mediation or coordination/security anchor. This enables the virtual disk, in one hand, to operate independently of any JANUS server, and in the other, to achieve better performance (as there is no server bottleneck). Moreover, since the cloud accounts are created and managed by the JANUS server, the clouds will never know the identity of the client. Note that, although JANUS have access to the data client stores

in the clouds, all the data is encrypted at the client-side with a key only it knows, ensuring that only the client has access to the original (unencrypted) data.
In the following, we detail the JANUS's main components.

7.2.1 Janus Server

This component acts as the front-end of the JANUS platform. It provides a user-interface in which clients describe their desired volume specifications, such as workload, requirements, constraints. Based on this information, JANUS finds the best possible storage profiles by defining the set of cloud services and storage techniques the virtual disk driver will use to store data in the clouds.
In this user-interface, the user can specify, namely, the read/write proportion of its workload, the maximum latency for accessing the data, the amount of data it intends to store, the maximum amount of money it wants to spend per month, the number of cloud services faults the system must tolerate, and restrictions regarding the locality of the data. The data locality restrictions are not only geographical (e.g., `only-in-europe` or `not-in-us`) but also in terms of cloud services to use (e.g., `not-in-amazon`) and diversity of locations and services (e.g., `not-same-continent` or `not-same-provider`). Moreover, clients are able to specify more than one locality restriction using logical operators (e.g., `only-in-europe AND not-in-belgium`).
JANUS server is composed of three modules as depicted in Figure 7.2.

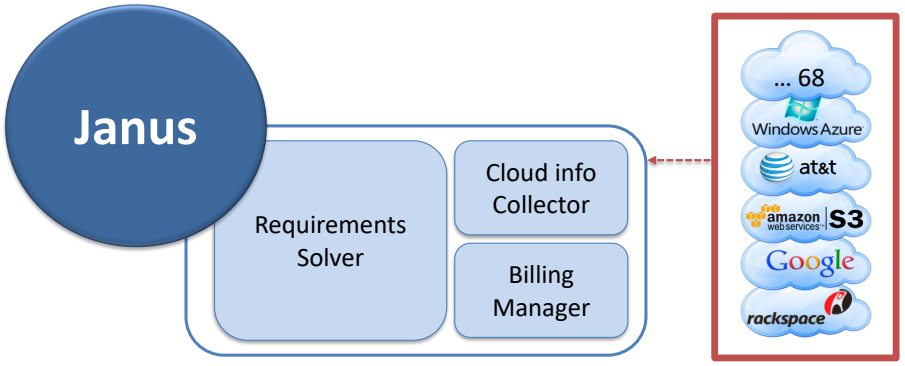


Figure 7.2: JANUS server modules.

These modules permit the server to find the best possible configuration to store data according to user-defined constraints.

7.2.1.1 Requirements Solver.

A module that, given the user volume specifications and the cloud informations (obtained by the cloud info collector module) finds the best storage profiles for that specifications. We give more details about this module in §7.2.3.

7.2.1.2 Cloud info collector.

This module is responsible for obtaining specific informations about all cloud services at all geographical locations the services support. Currently, the system aims to support 24 different cloud services, in 13 distinct providers, and in more than 68 geographical locations. The informations this module collects are the read, write and storage costs, the durability and availability SLAs, the consistency guarantees, and the security standards compliance of each cloud service location. They are collected periodically in order be kept up-to-date. The cloud services latency experienced by the client is not obtained by this module since it depends on the environment where the client's local machine is deployed. Besides, a latency test for each cloud service location is executed from its machine when the user executes the user-interface. Then, this information is attached to the volume specifications and

then sent to the server. The clouds information this module collects is used to feed the solver. Due, this module is also responsible for converting it to the format the solver is able to interpret after obtaining it.
This automatic collecting approach of such amount of clouds informations significantly reduces the management effort the system needs. This module is very important because gathering this information manually is time-consuming, non-scalable and error-prone.

7.2.1.3 Billing Manager.

A module that automatizes the creation and management of accounts in the different cloud storage providers used by the virtual disk to store the clients data. It is also used to maintain the billing information of each client, such as the amount of money spent in each cloud service for both reading and storing the data.
The cloud accounts' credentials are maintained secure in the server and are created only when needed. When a client chooses a storage profile generated by the solver, the server verifies each cloud service that will be used in that profile and creates accounts for the ones the client has no account already. The credentials of the services are sent to the client together with the virtual disk installer it downloads. To ensure the confidentiality of these credentials, they are encrypted before they are sent using a key shared between the client and the JANUS server.
Given this, when a client wants to obtain a storage profile adequate to some kind of data, it accesses the user-interface, defines its volume specifications and submits them to the server. When the JANUS server receives this request, it converts the high-level volume specifications to the format the solver interprets. Then, it provides this information to the solver, which uses it together with the information about the clouds (updated periodically by the cloud info collector module) to generate the optimal storage profiles for a volume. After letting the client choose the storage profile it wants, the server obtains the clouds credentials for the services used by that profile from the billing manager module. Finally, it creates the virtual disk installer, which includes a configuration file specifying the storage profile (cloud services and techniques) and the (encrypted) credentials for accessing the clouds.

7.2.2 Virtual Disk Driver

JANUS driver is installed at a client machine and it is responsible for interacting with the clouds to store and retrieve the systems' data. The cloud services and storage techniques it uses depend on the profile obtained from the JANUS server, which is generated according with the clients requirements. This driver exports various interfaces, being the client able to choose what is the best for its deployment. The envisioned interfaces are: local filesystem (based on FUSE and inotify), network filesystem (NFS and CIFS) and OpenStack's SWIFT.
In terms of data management, used techniques and data processing approach, our strategy is very similar to what is done in Charon [233]. The JANUS driver is able to use several techniques to ensure (1) the efficient use of cloud storage services both in terms of storage and latency, (2) the reliability of the data stored and (3) the performance of the local driver. Figure 7.3 depicts the operations the driver performs on the data to achieve the first two enumerated features.
The process starts by compressing the original file using the Huffman coding. After that, the compressed file is encrypted with a key maintained by the client. Since this step is done at the client-side and using a key only the client knows, the privacy of the data is ensured. When using multiple clouds, JANUS does not fully replicates data over them. Instead, it uses erasure codes techniques [266] to generate $q > f$ blocks, one for each cloud used, where q is the total number of clouds (a quorum) used for storing the file and f is the number of tolerated faults.[1] This technique significantly reduces the overhead associated with the storage of the data, decreasing the storage costs and the transferring latency at the same time. Finally, for performance reasons, the system sends one block for each cloud in parallel waiting only for the $n - f$ fastest responses.

[1]The exact value of q depends on the fault model considered, see Chapter 8.

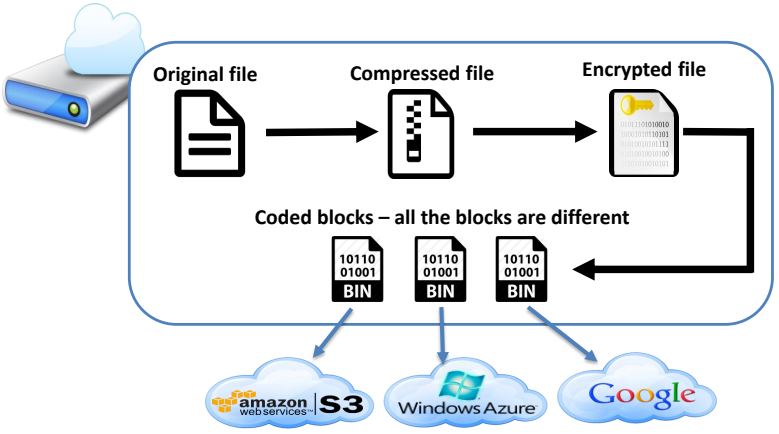


Figure 7.3: JANUS virtual disk driver data processing.

Notice that, for simplicity, in the figure we represent this process being performed over an entire file. However, in practice the file is first split into chunks of up to a specific size (16MB in our case) and the process is done over each of these chunks.
To improve the performance of the local driver, we *update data in the clouds in background*, use *caching*, and also use *prefetching* when reading big files from the clouds, just as in Mendes et al. [233].
As is the case with other systems [66, 161, 233, 314], the JANUS driver *separates data from metadata* by storing them in distinct objects in the clouds. Besides the modularity this strategy offers to the system, it allows us to easily change the guarantees provided by the system simply by switching the metadata service in use. A storage profile can tolerate both crash (CFT) or Byzantine (BFT) faults by the clouds, depending only on the storage protocol used to store the metadata object in the clouds (e.g., DepSky [65] for BFT and ICStore [54] for CFT). In JANUS, we use the new multi-writer protocols described in Chapter 8 to store metadata objects in the clouds in a consistent way, both for CFT and BFT operation modes.
Virtual disk driver can also be configured to optimize its behavior for application-specific workloads. Currently, we support the workload optimization for bioinformatics' applications [233] and disaster recover for two transactional database management systems, described in Chapter 9.

7.2.3 Solver

The Solver finds the best storage configuration for a given user-defined volume specification taking into account the information availbale about existing clouds. It is built using Prolog because this language makes the implementation of this kind of programs simpler than other solutions (such as SAT [230] or SMT [128] solvers), offering a good performance at the same time.
The cloud information is maintained in the JANUS solver knowledge base in the form of a set of Prolog *facts* and *relations*. The already mentioned clouds info collector module is the responsible for adding and updating this information in the knowledge base. Examples of a fact and a relation are `service(s3).` and `provider(s3, aws).`, respectively. The first example means that `s3` is a cloud service, and the second indicates that the provider of `s3` is `aws` (Amazon).
Besides these, the solver also maintains a set of static *rules* such as `same_provider(X, Y) :- provider(X, P), provider(Y, P).` This example permits to infer if two cloud services (`X` and `Y`) have the same provider `P`. These rules permit the solver to infer, for instance, how many clouds should be used (e.g., $3f + 1$ for BFT or $2f + 1$ for CFT volumes) or what are the best cloud services to meet the clients goals (e.g., cheaper or faster ones).
When the JANUS server receives a request to find the solutions for some give volume specifications, it first populates the solver knowledge base with the relations associating each cloud service location to

the latency experienced to each of them by the user's local machine. After that, it executes a *query* in the solver, by converting the specifications into Prolog *terms*. The query response contains the best storage configurations for the given volume specifications. The main informations each configuration indicates are: the cloud services to use (and their locations), the erasure codes parameters (which indicate the number of the blocks to be generated), the fault tolerant approach to use (single-cloud, CFT or BFT), and the storage/read cost per GB/month.

7.2.4 Query solving strategy.

The major challenge when solving the query to obtain the best storage configurations is finding the most adequate cloud services to use. This happens because the search space for the best combinations of cloud services could be very big. For instance, since we have 68 distinct service locations, the solver would need to compare the storage cost of $C(68, 10) > 2.9 \times 10^{11}$ combinations to find the most economic cloud service tolerating up to three Byzantine faults (requiring $n = 3f + 1$, with $f = 3$).
We minimize the computational requirements of this problem by dividing this process in two steps. First, we filter the cloud services according to the restrictions provided by the client. For example, if the client wants the data to be located only in Europe, we remove the services located in other locations. Note that this could be done with other restrictions (e.g., budget, latency).
The second step consists in obtaining only the m best solutions (m can also be specified by the client). This helps because in this way we do not need to search for all the possible solutions, only for the best ones. What we do is sort the possible cloud services according to a function, which allows us to generate the combinations in an ordered way. With this approach, when we find m solutions that satisfy all the combination restrictions (e.g., services in different providers), we are sure that these are the best ones. The function used to order the cloud storage services is generated by the solver after analyzing the volume specifications and it basically scores each service according to its interest regarding the client's goal. One example of a simple score function is the storage cost of the service, which give better scores to cheaper services.

7.3 Related Work

7.3.1 Distributed file systems.

The Janus driver adopts some ideas from existent file systems. It separates data and metadata like NASD [161], and performs updates in background like several peer-to-peer file systems [31, 204, 291]. With a serverless design, another related system is xFS [39], a network file system that stores all data and metadata at the client side.
A fundamental difference between these systems and Janus is that these systems do not explore the scalability and competitive prices of cloud storage, using instead the storage available on clients and servers.

7.3.2 Cloud-backed storage.

Commercial file synchronization services are popular solutions to backup personal data in the cloud [139, 166, 235]. However, these systems only use a single cloud and they are responsible for interacting with it, thus they have the control over the clients' data. Other interesting solutions are storage proxies, which transparently integrate local systems with cloud storage (e.g., [245, 254, 307]). The fundamental limitation of these systems is the fact that if the proxy fails, all the local systems connected to it will loose the access to the data.

7.3.3 Multi-cloud storage.

In the last years, many works have been proposing the use of multiple cloud providers to improve the integrity and availability of stored data. The idea was introduced in archival systems like RACS [30],

ICStore [54] and DepSky [65], which provide object storage (i.e., variable-size read/write registers) considering different fault models and unmodified storage clouds. However, the fact they provide only object storage hardens their integration with commonly used applications.
A hybrid approach is used by systems like Hybris [134] and SCFS [66], which employ unmodified cloud storage services used together with few computing nodes to store metadata and coordinate data access. In JANUS, the used server is totally independent of the data-plane of the system, meaning that it is used only to generate storage profiles. Due to this, the JANUS server does not affect the storage performance of the system, and thus can be deployed in a cheaper VM (with lower specifications).
A slightly different kind of works aggregate multiple file synchronization services in a single dependable service [172, 297], however they do not allow clients to define any storage requirement.
There are some works that permit clients to store their data according to some constraints [111, 233] or adapt their behavior in order to achieve some goal [264]. Charon [233] offers support to more than one storage repository (private, single cloud and cloud-of-clouds), which offer different guarantees in terms of reliability. In CYRUS [111] clients are able to specify the number of clouds in which data is stored (for privacy and fault tolerance) and from which the data is obtained when reading. FCFS [264] uses several cloud services and transparently moves the system's data across them in order to minimize costs. JANUS goes further since it does not obligate users to know what are the best clouds to use for their purposes (e.g., cost, latency). Instead, it creates the most appropriate storage profile for volumes matching each client goal.

7.4 Final Remarks

Users and companies have different types of data to store in the clouds, for which they have distinct storage constraints and are willing to spend different amounts of resources. JANUS is a platform to store data in the cloud according to user-defined requirements and constraints. This platform allows the clients to accommodate these diverse types of data in one service. JANUS offers storage solutions that can be compliant with distinct user-defined constraints such as reliability, data locality, budget, location/provider diversity, latency of accessing the data, security standards, etc. The system is able to store data both in a single and in multiple clouds, and in both cases the system ensures the privacy of the data. In the multi-clouds scenario, JANUS supports both CFT and BFT fault models.
We are currently implementing the platform and expect to have a first running version by the end of next semester (April, 2017).

Chapter 8 Exploring Key-Value Stores in Multi-Writer Byzantine-Resilient Cloud-of-Clouds Storage

Resilient register emulations on top of message passing systems are a cornerstone of fault-tolerant storage services. These emulations consider the provision of shared objects supporting read and write operations executed by a set of clients. In the traditional approach, these objects are implemented in a set of fail-prone servers (or replicas) that run some specific code for the emulation [43, 228, 173, 318, 133, 231, 86, 165, 227].
A less explored approach, dubbed data-centric, does not rely on servers that can run arbitrary code, but on passive replicas modeled as base objects that provide a constrained interface. These base objects can be as simple as a network-attached disk, a remote addressable memory, or a queue, or as complex as a transactional database, or a fully fledge cloud storage service. By combining these fail-prone base objects, one can build fault-tolerant services for storage, consensus, mutual exclusion, etc, using only client-side code, leading to arguably simpler and more manageable solutions.
The data-centric model has been discussed since the 90s [185], but the area gained visibility and practical appeal only with the emergence of network-attached disks technology [161]. In particular, several theoretical papers tried to establish lower bounds and impossibility results for implementing resilient read/write registers and consensus objects considering different types of fail-prone base objects (read/write registers [32, 152] vs. read-modify-write registers [106, 107]) under both crash and Byzantine fault models [29]. More recently, there has been a renascence of interest in data-centric algorithms for the cloud-of-clouds model. In this model, the base objects are cloud services (e.g., Amazon S3, Windows Azure Blob Storage) that offer interfaces similar to read/write registers or key-value stores (KVSs). These solutions ensure that the stored data is available even if a subset of cloud providers is unavailable or corrupts their copy of the data (events that do happen in practice [220]).
To the best of our knowledge, there are only two existing works for register emulation in the cloud-of-clouds model: DepSky [65], that tolerates Byzantine faults (e.g., data corruption or cloud misbehavior) on the providers but supports only a single-writer per data object, and Basescu et al. [54], that genuinely supports multiple writers, but tolerates only crash faults and does not support erasure codes.
In this chapter we present new register emulations on top of cloud storage services that support multiple concurrent writers (avoiding the need for expensive mutual exclusion algorithms [65]), tolerate Byzantine failures in base objects (minimizing the trust assumptions on cloud providers), and support erasure codes (decreasing the storage requirements significantly). In particular, we present three new multi-writer multi-reader (MWMR) regular register constructions:

1. an optimally-resilient register using full replication;
2. a register construction requiring more base objects, but achieving better storage-efficiency through the use of erasure codes;
3. another optimally-resilient register emulation that also supports erasure codes, but requires additional communication steps for writing.

These constructions are wait-free (operations terminate independently of other clients), uniform (they work with any number of clients), and can be adapted to provide atomic (instead of regular) semantics.

We achieve these results by exploring an often overlooked operation offered by KVSs – list – which returns the set of stored keys. The basic idea is that by embedding data integrity and authenticity proofs in the key associated with a written value, it is possible to use the list operation in multiple KVSs to detect concurrent writers and establish the current value of a register. Although KVSs are equivalent to registers in terms of synchronization power [82], the existence of the list operation in the interface of the former is crucial for our algorithms.
An additional benefit of supporting erasure codes when untrusted cloud providers are considered is that they can be substituted by a secret sharing primitive (e.g., [201]) or any privacy-aware encoding (e.g., [86, 266]), ensuring confidentiality of the stored data.
The three constructions we propose are described, proved correct, implemented and evaluated using real clouds (Amazon S3 [2], Microsoft Azure Storage [14], Rackspace Cloud Files [21], Google Storage [10] and Softlayer Cloud Storage [22]). Our experimental results show that these novel constructions provide interesting advantages both in terms of latency and storage costs.

8.1 Related Work

Existing fault-tolerant register emulations can be divided in two main groups depending on the nature of the fail-prone "storage blocks" that keep the stored data. The first group comprises the works that rely on servers capable of running part of the protocols [43, 228, 173, 318, 133], i.e., constructions that have both a client-side and a server-side of the protocol. Typically, in this kind of environment it is easier to provide robust solutions as servers can execute specific steps of the protocol atomically, independently of the number of clients accessing it.
In the second group we have the *data-centric* protocols [185, 32, 152, 106, 29]. This kind of solution considers a set of clients interacting with a set of passive servers with a constrained interface, modeled as shared memory objects (called base objects). The first work in this area was due to Jayanti, Chandra and Toueg [185], where the model was defined in terms of fail-prone shared memory objects. This work presented, among other wait-free emulations [174], a Byzantine fault-tolerant single-writer single-reader (SWSR) safe-register construction using $5f+1$ base objects to tolerate f faults. Further works tried to establish lower bounds and impossibility results for emulating registers tolerating different kinds of faults considering different types of base objects. For example, [32] and [152] used regular and/or atomic registers to implement crash-fault-tolerant MW and SW registers,[1] respectively, while [106] used read-modify-write objects to transform the SW register of [152] in a ranked register, a fundamental abstraction for implementing consensus. Abraham et al. [29] provided a Byzantine fault-tolerant SW register, which was latter used as a basis to implement consensus. The main limitation of these algorithms is that, although they are asymptotically efficient [32], the number of communication steps is still very large, and the required base objects are sometimes stronger than KVSs [106] or implement weak termination conditions [29].
More recently, there has been a renewed interest in data-centric algorithms for the cloud-of-clouds model [65, 54]. Here the base objects are cloud services offering interfaces similar to key-value stores. These solutions ensure that the stored data is available even if a subset of cloud providers is unavailable or corrupts their copy of the data. DepSky [65] provided a regular SW register construction that tolerates Byzantine faults by less than a third of the base objects, ensuring also the confidentiality of the stored data by using a secret sharing scheme [201]. However, to support multiple writers an expensive lock protocol must be executed to coordinate concurrent accesses. Another work in this line [54] provided a regular MW register that replicates the data by a majority of KVSs. Its main purpose was to reduce the required storage requirements. In this protocol, writers remove obsolete data synchronously, creating the need to store each version in two keys: a temporary key, that could be removed, and an eternal key, common for all writers and versions, that is never erased. In the best

[1] From now on we avoid characterizing the constructions about the number of readers, as all constructions discussed in the rest of the chapter support multiple readers (MR).

Table 8.1: Data-centric resilient register emulations. * can be extended to achieve atomic semantics.

Work	Fault model	Technique	Base Objects	Resilience	Semantics
Jayanti et al. [185]	Byzantine	replication	atomic registers	$5f+1$	SW safe
Gafni and Lamport [152]	crash	replication	atomic registers	$2f+1$	SW regular
Chockler and Malkhi [106]	crash	replication	rmw registers	$2f+1$	MW ranked
Abraham et al. [29]	Byzantine	replication	regular registers	$3f+1$	SW regular
	Byzantine	replication	regular registers	$3f+1$	SW safe
Aguilera and Gafni [32]	crash	replication	atomic registers	$2f+1$	MW atomic
Bessani et al. [65]	Byzantine	erasure code	regular registers	$3f+1$	SW regular
Basescu et al. [54]	crash	replication	atomic KVSs	$2f+1$	MW regular*
This work	Byzantine	replication	atomic KVSs	$3f+1$	MW regular*
	Byzantine	erasure code	atomic KVSs	$4f+1$	MW regular*
	Byzantine	erasure code	atomic KVSs	$3f+1$	MW regular*

case, the algorithm requires a storage space of $2 \times S \times n$, where S is the size of the data and n is the number of KVSs.
Using registers or KVSs as base objects in the data-centric model makes it more challenging to implement dependable register emulations, as general replicas have more synchronization power than such objects [82]. The three new algorithms presented in this chapter advance the state of the art by supporting multiple writers and erasure-coded data in the data-centric Byzantine model, using a rather weak base object – a KVS. Two of these constructions have optimal resilience, as they require $3f+1$ base objects to tolerate f Byzantine faults in an asynchronous system (with confirmable writes) [231]. Table 8.1 summarizes the discussed data-centric constructions.

8.2 System Model

8.2.1 Register Emulation

We consider an *asynchronous* system comprised by a finite set of clients and n cloud storage providers, which provide a KVS interface. We refer to clients as *processes* and to cloud storage providers as *base objects*. Each process has a unique identifier from an infinite set named *IDs*, while the base objects are numbered from 0 to $n-1$.
We aim to provide MW-register abstractions on top of n base objects. This kind of abstraction offers an interface specification composed by two *operations*: **write(v)** and **read()**. The sequential specification of a register requires that a read operation returns the last value written, or $\perp$ if no write has ever been executed. Processes interacting with registers can be either *writers* or *readers*.
A process operation starts by an *invoke* action on the register, and ends with a *response*. An operation *completes* when the process receives the response. An operation o_1 *precedes* another operation o_2 (and o_2 *follows* o_1) if it completes before the invocation of o_2. Operations with no precedence relation, are called *concurrent*.
Unless stated otherwise, the register implementations should be *wait-free* [174], i.e., the operation invocations should complete in a finite number of internal steps. Moreover, we want to provide *uniform* implementations, i.e., implementations that do not rely on the number of processes, allowing processes to not know each other initially.
We provide two register abstraction semantics, *regular* and *atomic*, which differ mainly in the way they deal with concurrent accesses [209]. In a regular register it is only guaranteed that different read operations agree on the order of preceding write operations. Any read operation overlapping a write operation may return the value being written or the preceding value. An atomic register employs a stronger consistency notion than regular semantics. It stipulates that it should be possible to place each operation at a singular point (linearization point) between its invocation and response. Basically,

after a read operation completes, a following read must return at least the version returned in the preceding read, even in the presence of concurrent writes.

8.2.2 Threat Model

Up to f out-of n base objects can be subject to NR-arbitrary failures [185], which are also known as Byzantine failures. The behavior of such objects can be unrestricted: they may not respond to an invocation, and if they do, the content of the response may be arbitrary. Unless stated otherwise, readers may also be subject to Byzantine failures. Writers can only fail by crashing, because even if the protocol tolerates Byzantine writers, they may always store arbitrary values or overwrite data on the register. Processes and base objects are said to be *correct* if they do not fail.
For cryptography, we assume that each writer has a private key K_r to sign some of the information stored on the base objects. These signatures can be verified by any process in the system through the corresponding public key K_u. Moreover, we also assume the existence of a collision-resistant cryptographic hash function to ensure integrity. There might be multiple writer keys as long as readers can access their public counterparts.

8.2.3 Key-Value Store Specification

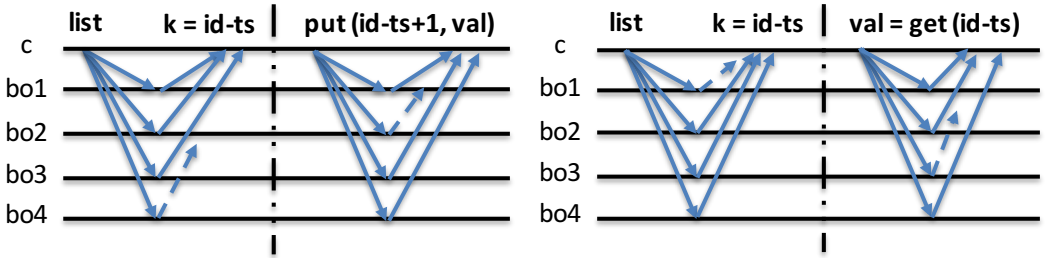
Current cloud storage service providers offer a key-value store (KVS) interface for customers to store and retrieve their data. In this context, KVSs act as passive servers where it is impossible to run any client code, forcing the implementations to be *data-centric*. Being associative arrays, they export four well defined operations to interact with a collection of $\langle key, value \rangle$ pairs, where any *key* can have only one value associated. The size of stored values are expected to be much larger than the size of the associated keys. The four operations are: (1) **put(**k**,** v**)**, (2) **get(**k**)**, (3) **list()**, and (4) **remove(**k**)**. The first operation associates a key k with the value v, returning *ack* if successful and *ERROR* otherwise; the second retrieves the value associated with a key k, or *ERROR* if the key does not exist; the third returns an array with all the keys in the collection, or [] if there are no keys in the collection; and the last operation disassociates a key k from its value, releasing storage space and the key itself, returning an *ack* if successful and *ERROR* otherwise. Finally, we assume that individual KVS's operations are atomic and wait-free.

8.3 Multi-Writer Constructions

In this section we describe the three MW-regular register implementations. Before discussing the algorithms in detail (§8.3.3 to §8.3.6), we present an overview of the general structure of the protocols (§8.3.1) and describe the main techniques employed in their construction (§8.3.2). The correctness proofs of the protocols are presented in §8.4.

8.3.1 Overview

Our three MW-regular protocols differ in the storage technique employed (replication or erasure code), the number of base objects required ($3f + 1$ or $4f + 1$), and the number of sequential base object accesses (two or three steps). Despite of that, the general structure of all protocols is similar to the one illustrated in Figure 11.1.
In the write operation, the client first lists a quorum of base objects (KVSs) in order to find the key encoding the most recent version written in the system, and then puts the value been written associated with a unique key encoding a new (incremented) version in a quorum. The read operation requires finding the most recent version of the object (as in the first phase of the write operation), and then retrieving the value associated with that key.
Notice that our approach considers that each written value requires a new key-value pair in the KVSs. However, it is impossible to implement wait-free data-centric MR-regular register emulations without



(a) WRITE. (b) READ.

Figure 8.1: MW-regular register protocols general structure.

using at least one "data element" per written version if the base objects do not provide conditional update primitives (similar to Compare-and-Swap) [54, 107]. Therefore, any practical implementation of these algorithms must consider some form of garbage collection, as discussed in §8.5.2.

8.3.2 Protocols Mechanisms

Our algorithms use a set of mechanisms that are crucial for achieving Byzantine fault tolerance, MW semantics and storage efficiency. These mechanisms are discussed bellow.

8.3.2.1 Byzantine Quorum Systems

Our protocols employ dissemination and masking Byzantine quorum systems to tolerate up to f Byzantine faults [228]. Dissemination quorum systems consider quorums of $q = \lceil \frac{n+f+1}{2} \rceil$ base objects, requiring thus a total of $n > 3f$ base objects in the system. This ensures each two quorums intersect in at least $f+1$ objects (one correct). Masking quorum systems require quorums of size $q = \lceil \frac{n+2f+1}{2} \rceil$ and a total of $n > 4f$ base objects, ensuring thus quorum intersections with at least $2f+1$ base objects (a majority of correct ones).

8.3.2.2 Multi-Writer Semantics

We use the **list** operation offered by KVSs to design MW uniform implementations. This operation is very important as it allows us to discover new versions written by unknown clients. With this, the key idea of our protocols is making each writer to write in its own abstract register in a similar way to what is done in traditional transformation of SW to MW registers [209]. We achieve this by putting the client unique id on each key alongside with a timestamp ts, resulting in the pair $\langle ts, id \rangle$, which represents a *version*. This approach ensures that clients writing new versions of the data never overwrite versions of each other.

8.3.2.3 Object integrity and authenticity

We call the pair $\langle data\ key, data\ value \rangle$ an *object*. In our algorithms, the *data key*[2] is represented by a tuple $\langle ts, id, h \rangle_s$, where $\langle ts, id \rangle$ is the version, h is a cryptographic hash of the *data value* associated with this key, and s is a signature of $\langle ts, id, h \rangle$ (there is a slight difference in the protocol of §8.3.6, as will be discussed later). Having all this information on the data key allows us to validate the integrity and authenticity of the version (obtained through the **list** operation) before reading the data associated with it. Furthermore, if some version has a valid signature we call it *valid*. A data value is said to be valid if its hash matches the hash present in a valid key (this can only be proved after reading the value associated with the key). Consequently, an object is valid if both the version and the value are valid.

[2]For the rest of this chapter we may refer to this only as key.

Algorithm 1: Auxiliary functions.

```
1  Function listQuorum() begin
2      L[0..n − 1] ←⊥;
3      concurrently for 0 ≤ i ≤ n − 1 do
4          L[i] ← list_i;
5      wait until |{i : L[i] ≠⊥}| ≥ q;
6      return L;
7  Function writeQuorum(data_key, value) begin
8      ACK[0..n − 1] ←⊥;
9      concurrently for 0 ≤ i ≤ n − 1 do
10         ACK[i] ← put(data_key, value[i])_i;
11     wait until |{i : ACK[i] = true}| ≥ q;
12 Function maxValidVersion(L) begin
13     return ⟨vr, h⟩_s ∈ ⋃_{i=0}^{n−1} L[i] : verify(s, K_u) ∧ ∄ ⟨vr′, h′⟩_{s′} ∈ ⋃_{i=0}^{n−1} L[i] : vr′ > vr ∧ verify(s′, K_u)) ;
```

8.3.2.4 Erasure codes

Two of our protocols employ erasure codes [266] to decrease the storage overhead associated with full replication. This technique generates n coded blocks, one for each base object, from which any $m < q$ base objects blocks can reconstruct the data. Concretely, in our protocols we use $m = f + 1$.
Notice that this formulation of coded storage can also be used to ensure confidentiality of the stored data, by combining the erasure code with a secret sharing scheme [201], in the same way it was done in DepSky [65].

8.3.3 Pseudo Code Notation and Auxiliary Functions

We use the '+' operator to represent the concatenation of strings and the '.' operator to access data key fields. We represent the parallelization of base object calls with the tag **concurrently**. Moreover, we assume the existence of a set of functions: (1) *H(v)* generates the cryptographic hash of v; (2) *encode(v, n, m)* encodes v in n blocks from which any m are sufficient to recover it; (3) *decode(bks, n, m, h)* recovers a value v by decoding any subset of m out-of n blocks from the array *bks* if $H(v) = h$, returning $\perp$ otherwise; (4) *sign(info, K_r)* signs *info* with the private key K_r, returning the resulting signature s; (5) *verify(s, K_u)* verifies the authenticity of signature s using a public key K_u.
Besides these cryptographic and coding functions, our algorithms employ three auxiliary functions, described in Algorithm 1. The first function, *listQuorum* (Lines 1-6), is used to (concurrently) list the keys present in a quorum of KVSs. It returns an array L with the result of the list operation in at least q KVSs.
The *writeQuorum*(*data_key*, *value*) function (Lines 7-11) is used for clients to write data in a quorum of KVSs. The key *data_key* is equal in all base objects, but the value *value*[i] may be different in each base object, to accommodate erasure-coded storage. When at least q successful put operations are performed, the loop is interrupted.
The last function, *maxValidVersion*(L) finds the maximum version number correctly signed on an array L containing up to n KVS' list results (possibly returned from *listQuorum* function), returning 0 (zero) if no valid version is found.

8.3.4 Two-Step Full Replication Construction

Our first Byzantine fault-tolerant MW-regular register construction employs full replication, storing thus the whole value in each base object. The algorithm is optimally resilient as it employs a dissemination quorum system [228]. Algorithm 2 presents the write and read procedures for the construction.

Algorithm 2: Regular Byzantine Full Replication (FR) MW register ($n > 3f$) for client c.

```
1  Procedure FR-write(value) begin
2      L ← listQuorum();
3      max ← maxValidVersion(L);
4      new_key ← ⟨max.ts + 1, c, H(value)⟩;
5      data_key ← new_key + sign(new_key, K_r);
6      v[0..n − 1] ← value;
7      writeQuorum(data_key, v);
8  Procedure FR-read() begin
9      L ← listQuorum();
10     repeat
11         data_key ← maxValidVersion(L);
12         d[0..n − 1] ←⊥;
13         concurrently for 0 ≤ i ≤ n − 1 do
14             value_i ← get(data_key)_i;
15             if H(value_i) = data_key.hash then
16                 d[i] ← value_i;
17             else
18                 d[i] ← ERROR;
19         wait until (∃i : d[i] ≠⊥ ∧d[i] ≠ ERROR) ∨ (|{i : d[i] ≠⊥}| ≥ q);
20         ∀i ∈ {0, n − 1} : L[i] ← L[i] \ {data_key};
21     until ∃i : d[i] ≠⊥ ∧d[i] ≠ ERROR;
22     return d[i];
```

Processes perform write operations using the procedure **FR-write** (Lines 1–7). The protocol starts by listing a quorum of base objects (Line 2). Then, it finds the maximum version available with a valid signature in the result using the function $maxValidVersion(L)$ (Line 3), and creates the new data key by concatenating a new unique version, and the hash of the value together with the signature of these fields (Lines 4–5). Lastly, it uses the *writeQuorum* function to write the data to the base objects (Lines 7).

The read operation is represented in the **FR-read** procedure (Lines 8–22). As in the write operation, it starts by listing a quorum of base objects. Then the reader enters in a loop until it reads a valid value (Line 10–21). First, it gets the maximum valid version listed (Line 11), and then it triggers n parallel threads to read that version from different KVSs. Next, it waits either for a valid value, which is immediately returned, or for a quorum of q responses (Line 19). The only way the loop terminates due to the second condition is if it is trying to read a version being written concurrently with the current operation, i.e., a version that is not yet available in a quorum. This is possible if the first q base objects to respond do not have the maximum version available yet. When this happens, the version is removed from the result of the **list** operation (Line 20), and another iteration of the outer loop is executed to fetch a smaller version. Notice that a version that belongs to a complete write can always be retrieved from the inner loop due to the existence of at least one correct base object in the intersection between Byzantine quorums.

Without concurrency, the protocol requires one round of **list** and one of **put** for writing, and one round of **list** and one of **get** for reading. In fact, it is impossible to implement a MW register with less base object calls since for writing and reading we always need to use at least one round of **put** and **get** operations, respectively, and to find the maximum version available we can only use **list** or **get** to retrieve that information from the base objects.

8.3.5 Two-Step Erasure Code Construction

Differently from the protocol described in the previous section, which employs full replication with a storage requirement of $q \times S$ wherein S is the size of the object, in our second Byzantine fault-tolerant MW-regular register emulation we use storage-optimal erasure codes. Since the erasure code

Algorithm 3: Regular Byzantine Erasure-Coded (EC) MW register ($n > 4f$) for client c.

```
1  Procedure EC-write(value) begin
2      L ← listQuorum();
3      max ← maxValidVersion(L);
4      new_key ← ⟨max.ts + 1, c, H(value)⟩;
5      data_key ← new_key + sign(new_key, K_r);
6      v[0..n − 1] ← encode(value, n, f + 1);
7      writeQuorum(data_key, v);
8  Procedure EC-read() begin
9      L ← listQuorum();
10     foreach ver ∈ L : #_L(ver) < f + 1 do
11         ∀i ∈ {0, n − 1} : L[i] ← L[i] \ {ver};
12     repeat
13         data_key ← maxValidVersion(L);
14         data ←⊥;
15         concurrently for 0 ≤ i ≤ n − 1 do
16             d[i] ← get(data_key)_i;
17             if data =⊥ then
18                 data ← decode(d, n, f + 1, data_key.hash);
19         wait until data ≠⊥ ∨ |{i : d[i] ≠⊥}| ≥ q;
20         ∀i ∈ {0, n − 1} : L[i] ← L[i] \ {data_key};
21     until data ≠⊥ ∧ data ≠ ERROR;
22     return data;
```

we use [266] generates n coded blocks, each with $\frac{1}{f+1}$ of the size of the data, the storage requirement is reduced to $q \times \frac{S}{f+1}$.

The main consequence of storing different blocks in different base objects for the same version, is the number of base objects accessed in a dissemination quorum systems is not enough to construct a wait-free Byzantine fault-tolerant MW-regular register. This happens because the intersection between dissemination quorums contains only $f + 1$ base objects, meaning that when reading the version associated with the last complete write operation, the quorum accessed may contain only 1 valid response (f can be faulty). This is fine for full replication as a single updated and correct value is enough to complete a read operation. However, it may lead to a violation of the regular semantics when erasure codes are employed since we now need at least $f + 1$ encoded blocks to reconstruct the last written value.

To overcome this issue we use Byzantine masking quorum systems [228], where the quorums intersect in at least $2f + 1$ base objects. Despite the increase in the number of base objects ($n > 4f$), the storage requirement is still significantly reduced when compared with the previous protocol. As an example, for $f = 1$, this protocol has a storage overhead of 100% (a quorum of four objects with coded blocks of half of the original data size) while in the previous protocol the overhead is 200% (a quorum of three objects with a full copy of the data on each of them).

Algorithm 3 presents the details about this protocol. The **EC-write** procedure is similar to the write procedure of Algorithm 2. The only difference is the use of erasure codes to store the data. Instead of full replicating the data, it uses the *writeQuorum* function to spread the generated erasure-coded blocks through the base objects in such a way that each one of them will store a different block (Lines 6–7). Notice that the hash on the data key is generated over the full copy of data and not over each of the coded blocks.

The read procedure **EC-read** is also similar to the read protocol described in §8.3.4, but with two important differences. First we remove from L the versions we consider impossible to read (Lines 10–11), i.e., versions that appear in less than $f + 1$ responses obtained from different KVSs. Second, instead of waiting for one valid response in the inner loop, we wait until we can reconstruct the data or for a quorum of responses. Again, the only way the loop terminates through the second condition

Algorithm 4: Regular Byzantine Erasure-Coded (EC) MW register ($n > 4f$) for client c.

```
1  Procedure 3S-write(value) begin
2      L ← listQuorum();
3      max ← maxValidVersion(L);
4      data_key ← ⟨max.ts + 1, c⟩;
5      proof_info ← "PoW" + ⟨max.ts + 1, c, H(value)⟩;
6      proof_key ← proof_info + sign(proof_info, K_r) ;
7      v[0..n − 1] ← encode(value, n, f + 1);
8      writeQuorum(data_key, v);
9      v[0..n − 1] ← ∅;
10     writeQuorum(proof_key, v);
11 Procedure 3S-read() begin
12     L ← listQuorum();
13     proof_key ← maxValidVersion(L);
14     data_key ← ⟨proof_key.ts, proof_key.id⟩;
15     data ←⊥;
16     concurrently for 0 ≤ i ≤ n − 1 do
17         d[i] ← get(data_key)_i;
18         if data =⊥ then
19             data ← decode(d, n, f + 1, data_key.hash);
20     wait until data ≠⊥;
21     return data;
```

is if we are trying to read a concurrent version. For reconstructing the original data, every time a new response arrives we try to decode the blocks and verify the integrity of the obtained data (Line 18). Notice that the integrity is verified inside the *decode* function. A version associated with a complete write can always be successfully decoded because any accessed quorum will provide at least $f+1$ valid blocks for decoding this version's value. As soon as the integrity is verified, the outer loop stops and the value is returned (Lines 21–22).

8.3.6 Three-Step Erasure Code Construction

Our last construction implements a Byzantine-resilient MW-regular register using erasure codes and dissemination quorums, being thus both storage-efficient and optimally-resilient. We achieve this by storing in each base object two objects per version instead of one.

The first object we call *data object* and is used to store the encoded data blocks. The second one, named *proof object*, is a zero-byte object used to prove that a given data object is already available in a quorum of base objects (similar to what is done in previous works [133, 65]). The key of the data object is composed only by the version, i.e., the tuple $\langle ts, id \rangle$. In turn, the key of the proof object is composed by the string $\langle \text{"PoW"}, ts, id, h \rangle_s$, in which h is the hash of the full copy of data and s is a signature of $\langle \text{"PoW"}, ts, id, h \rangle$.

Algorithm 4 presents the protocol. The write procedure, called **3S-write**, starts by listing the proof objects from a quorum of base objects (Line 2). Then it finds the maximum valid version between the proof objects. For simplicity, this algorithm is using the same function $maxValidVersion(L)$ as the previous protocols, but here we are only focusing in proof objects. Next it creates the new data key and the new proof key to be written (Lines 4–6). Then it writes the data object in a quorum (ensuring that different base objects will store different coded blocks) and, after that, it writes the proof object (Lines 7-10). This sequence of actions ensures that when a valid proof object is found in at least one base object, the corresponding data object is already available in a quorum of base objects.

The **3S-read** procedure is used for reading. The idea is to list proof objects from a quorum, find the maximum valid version among them, and read the data object associated with that proof object. Notice that to read the data we do not need to wait for a quorum of responses as it is enough to have $m = f + 1$ valid blocks to decode the value (Lines 18–19). This holds because, differently from the

two previous algorithms, here we are sure that the data values with a version matching the maximum version found in valid proof objects is already stored in a quorum of base objects.
As explained before, this protocol works with only $3f + 1$ base objects. This is done without adding any extra call to the base objects in the read operation, which still needs only two rounds of accesses, one for **list** and one for **get**. However, for writing, one more round of **put** is needed (to replicate the proof object). This trade-off is actually profitable in a cloud-of-clouds environment since the monetary costs of storing erasure-coded blocks in extra clouds is much larger than sending zero-byte objects to the clouds we use.

8.4 Correctness

This section presents the correctness proofs of the protocols of Algs. 2, 3 and 4. We start by proving that the auxiliary functions used by the protocols (presented in Alg. 1) are wait-free.

Lemma 2. *Every correct process completes the execution of listQuorum and writeQuorum in finite time.*

Proof. Both algorithms are used by all protocols. This means that they use both dissemination and masking quorums. Since they send n requests, one for each base object, and at most f base objects are allowed to be faulty, a quorum of responses will always be received as $q \leq n - f$. Consequently, both algorithms return in finite time. □

8.4.1 Two-Step Algorithms Proof

In the following we prove the correctness of Algs. 2 and 3, as they follow the same rationale, although employing different Byzantine quorum systems. For these proofs, we denote by L the output of the *listQuorum* executed in the beginning of **FR-read** and **EC-read** procedures. Moreover we define m as the number of required responses to obtain the requested value. Notice that $m = 1$ for full replication and $m = f + 1$ for erasure-coded data. The next lemmas state that L contains a version that respects MW-regular semantics.

Lemma 3. *A value associated with a complete write operation is always found in L, and can be retrieved from the base objects.*

Proof. Since we do not consider malicious writers, we know that they only write valid objects in a quorum q. Furthermore, we know that when listing or reading a dissemination (resp. masking) Byzantine quorum of base objects we will also access $f + 1$ (resp. $2f + 1$) objects where the last complete write was executed, as by definition quorums intersect by this amount of objects. Using this fact, the lemma can then be reduced to prove that such intersection will contain m correct base objects, which will provide the last version written. This is indeed the case as $m = 1$ in full replication (intersection of $f + 1$ – at least one correct) and $m = f + 1$ with erasure coded data (intersection of $2f + 1$ – at least $f + 1$ correct). □

Lemma 4. *The maximum version found on L corresponds to the last complete write operation, or to a concurrent one.*

Proof. L only contains valid versions that were already stored by a writer, otherwise it would be impossible to find them. Since we do not consider malicious writers, we know they will follow the protocol, for example, by incrementing the maximum ts found and has not lied about his id when creating the pair $\langle ts + 1, c \rangle$. Therefore, each writer always writes a version larger than the maximum version found, respecting thus partial order.
According to Lemma 3, a version whose write is complete is always found in L. Consequently, we can claim that, without concurrency, the maximum version in L belongs to the last complete write operation. Furthermore, if there are any concurrent write operation being executed, it may appear as

the maximum version found in L, as its version is surely greater then the version of the last complete write. □
These lemmas allow us to prove that both protocols (Algs. 2 and 3) respect the specification of a multi-writer multi-reader regular register and are wait-free.

Theorem 2. *A FR-read (resp. EC-read) operation running concurrently with zero or more FR-write (resp. EC-write) operations will return the value associated with the last complete write or one of the values being written.*

Proof. Both read procedures start by calling *listQuorum*. By Lemma 4 we know that the maximum version found in L belongs to the last complete write operation or to a concurrent one. Independently of the case, the procedures try to read it. If the version belongs to a concurrent write it may not be retrieved. In this case the algorithms exclude it and fetch the new maximum valid version listed (see loop in Lines 10–21 and 12–21, in Algorithms 2 and 3, respectively). However, according to Lemma 3, if no concurrent version can be read, we know that the value associated with the last complete version is always retrieved. This proves that both protocols respect regular semantics. □

Theorem 3. *The FR-write, EC-write, FR-read and EC-read procedures satisfy wait-freedom.*

Proof. The **FR-write** and **EC-write** procedures, besides executing local computation steps, call the functions of Algorithm 1. Since these algorithms are wait-free (Lemma 2), the write protocols are also wait-free.
Both read operations start by calling the *listQuorum*, which is wait-free (Lemma 2). After that, the algorithms enter in a loop that only terminates after finding a valid value to return. By Lemma 3 we know that a value associated with a complete version is always found in L, and that this version can be retrieved. Then the number of iterations of this loop is bounded by the number of writes being executed concurrently that can be seen in L, but whose the value cannot be retrieved. Since after failing a read we try a smaller version, the algorithms will eventually try to fetch the value written in the last complete write. Consequently, the read procedures terminate in finite time. □

8.4.2 Three-Step Algorithm Proof

We now sketch the correctness proof of Algorithm 4. The complete proofs are very similar to the ones presented before for the Algs. 2 and 3. The main difference here is the existence of the proof object used to prove that the data object associated with it is already stored in a dissemination quorum. The following lemmas state the properties of this object.

Lemma 5. *The value associated with every valid proof object in L can be retrieved from at least $f+1$ base objects.*

Proof. Each valid proof object found in L was previously written by a correct writer. In turn, since we do not consider malicious writers, each writer only replicates the proof object after storing its associated data object in at least $q = \lceil \frac{n+f+1}{2} \rceil$ base objects. Therefore, we know that the data object associated with a valid proof object in L is available in at least $q - f \geq f + 1$ base objects. This means that there will be enough data objects to reconstruct the original value. □

Lemma 6. *The maximum valid version found among the proof objects observed in L corresponds to the last complete write, or to a concurrent one.*

Proof. This can be proved following the same rationale of Lemma 4. Writers are considered correct and therefore they calculate new versions correctly, i.e., they find the maximum valid version on a quorum of proof objects and increment it ensuring that each new version has a greater *version* number. Furthermore, an **3S-write** operation is considered complete only after it writes the proof object to a dissemination quorum. Since we know that the intersection of any two dissemination quorums contains at least $f + 1$ base objects, the proof object associated with the last complete write can always be

found. This proves that, without concurrency, the maximum version found corresponds to the last complete write operation. If some concurrent writes are being executed (Line 10), they may be seen as the maximum version because they surely have a greater version than the last complete write. □
Using these lemmas we are now able to prove that Algorithm 4 respects multi-writer multi-reader regular register semantics and wait-freedom.

Theorem 4. *A 3S-read operation running concurrently with zero or more 3S-write operations will return the value associated with the last complete write or one of the values being written.*

Proof. According to Lemma 5, the value associated with each valid proof object in L can always be read from the base objects. The protocol reads the value associated with the maximum valid version found in L, and we know by Lemma 6 that this version is associated either with the last complete write or to one of the values being written. Therefore, the value returned by **3S-read** belongs to the last complete write or to a concurrent one. □

Theorem 5. *The 3S-read and 3S-write procedures satisfy wait freedom.*

Proof. The write procedure invokes *listQuorum* once, to obtain L, and *writeQuorum* twice, one to write the data blocks and another to write the proof objects. According to Lemma 2, these two operations terminate in finite time, and thus **3S-write** always terminates as well.
The read procedure also satisfies wait freedom due to Lemma 5: a value associated with a valid proof object is always available for read. □

8.5 Protocols Extensions

This section presents a discussion of how the protocols presented in this chapter can be modified to offer atomic semantics [209], and what are the possible solutions to garbage collect obsolete data versions.

8.5.1 Atomicity

There are many known techniques to transform regular registers in atomic ones. Most of them require servers running part of the protocol [86, 231], which is impossible to implement with our base objects. Fortunately, the simplest transformation can be used in data-centric algorithms. This technique consists in forcing readers to *write-back* the data they read to ensure this data will be available in a quorum when the read completes [165, 227, 54].
Our three constructions could implement this technique by invoking *writeQuorum* to write the read value before returning it. However, writing back read values in our first two protocols may carry out performance issues as the stored data size might be non-negligible. In turn, employing the same write-back technique in our last protocol (Algorithm 4) does not have this limitation, as a reader would only need to write-back the zero-byte proof object (see §8.3.6). Hence, the performance effect of using this technique in the read procedure is independent of the size of the data being read. A final concern about using write-backs to achieve atomicity is the that we would have to assume that readers may only fail by crash, otherwise they may write bogus values in the base objects. In the regular constructions this is not required as we do not need to give write permissions to readers.

8.5.2 Garbage Collection

Existing solutions. Register emulations that employ versioning must use a garbage collection protocol to remove obsolete versions, otherwise an unbounded amount of storage is required. DepSky [65] provides a garbage collection protocol that is triggered periodically to remove older versions from the system. Although practical in many scenarios, this solution is vulnerable to the *garbage collection racing problem* [54, 318]. This problem happens when a client is reading a version that had became

obsolete due to a concurrent write, and removed by a concurrent execution of the garbage collection protocol, making it impossible for a reader to obtain the value associated with it.
To the best of our knowledge, there are only two works that solve this problem. The solution of [318] makes readers announce the version they are going to read, preventing the garbage collector from deleting it. Unfortunately, this solution cannot be directly applied in the data-centric model since it requires servers capable of running parts of the algorithm. Another solution was proposed in [54]. In this protocol each writer stores the value in a *temporary* key, which can be garbage collected by other writers, and also in an *eternal* key, that is never deleted. This approach allows readers to obtain the value from the eternal key when the temporary key is erased by concurrent writers. A solution like this can be applied to our first protocol (see §8.3.4), which employs full replication. Yet, it does not work with erasure-coded data. The reason is the eternal key is overwritten whenever a write operation occurs, and since several writers can operate simultaneously, the eternal key in different base objects may end up with blocks belonging to different versions. Therefore, it might lead to the impossibility of getting $f + 1$ blocks of the same version to reconstruct the original value.
Adapting the solutions to our protocols. All existing solutions for garbage collection can be adapted to the protocols discussed in §8.3. The approach of deleting obsolete versions asynchronously by a thread running in background can be naturally integrated to our protocols. This thread can be triggered by the clients at the end of the write operations, making each client responsible for removing its obsolete data.
Since we do not rely on server-side code for our protocols, devising a solution where readers announce the version they are about to read (by writing an object with that information to a quorum of base objects) would require substantial changes in our system model. More specifically, to ensure wait-freedom for read operations, only objects with versions lower than the ones announced can be garbage collected. This solution may not tolerate the crash of the readers – if a reader crashes without removing its announcement, larger versions than the one it announced will never be removed. It is possible to add an expiration time to the announcement to avoid this. Yet, this would still require changes in the system model to add synchrony assumptions for the expiration time to (eventually) hold, and not consider Byzantine readers (that could block garbage collection by announcing the intention to read all versions).
Using the eternal key approach together with erasure codes significantly increase the storage requirements of our algorithms. The idea is to make each writer not only store the coded blocks into temporary keys, but also replicate full copies of the original data in eternal keys. This approach may lead to a decrease in the write performance (related with an extra write of a full copy of the data per base object) and an increase of $n \times S$ in each protocol storage requirements.
Discussion. The three proposed solutions explore different points in the design space of data-centric storage protocols. In the first approach, we do not really solve the garbage collection racing problem. The second solution requires a stronger system model and additional base object accesses in the read operation. The third solution increases the storage requirements and reduces the write performance as writers have to write not only the coded blocks, but also full copies of the data.
We argue that most applications would prefer to have better performance and low storage requirements, at the cost of eventually repeating failed reads. Therefore, we chose to support the asynchronous garbage collection triggered periodically (for example hourly, daily or even when a given number of versions has been written), as done in DepSky [65].

8.6 Evaluation

This section presents an evaluation of our three new protocols, comparing them with two previous constructions targeting the cloud-of-clouds model [54, 65].

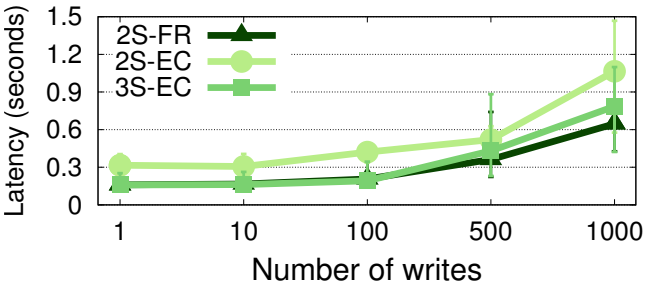


Figure 8.2: Average latency and std. deviation of *listQuorum* for different number of stored keys.

8.6.1 Setup and Methodology

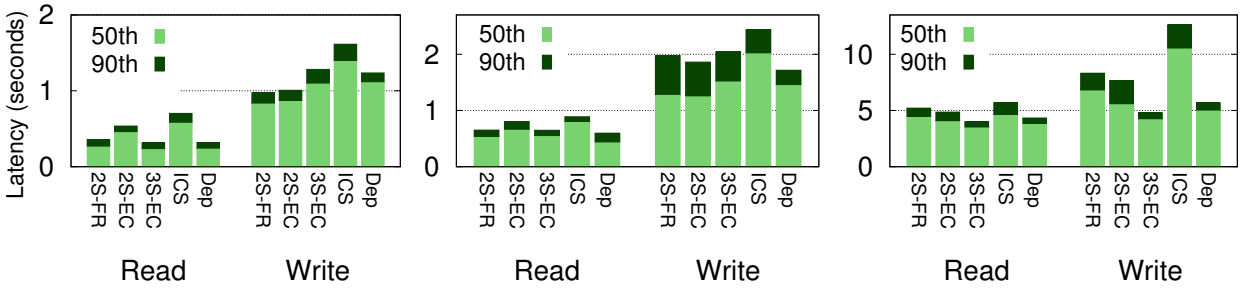
The evaluation was done using a Dell Power Edge R410 machine equipped with two Intel Xeon E5520 (quad-core, HT, 2.27Ghz), and 32GB of RAM. This machine was running an Ubuntu Server Precise Pangolin operative system (12.04 LTS, 64-bits, kernel 3.5.0-23-generic), and Java 1.8.0_67 (64-bits). Furthermore, we compare our protocols with the MW-regular register of [54], which we call ICS, and the SW-regular register of DepSky [65]. The protocols proposed in this chapter were implemented in Java using the APIs provided by real storage clouds. We used the DepSky implementation available online [8]. However, since there is no available implementation of ICS, we implemented it using the same framework we used for our protocols. All the code used in our experiments is available on the web [15].
All experiments consider $f = 1$ and the presented results are an average of 1000 executions of the same operation, employing garbage collection after every 100 measurements. The storage clouds used were Amazon S3 [2], Google Storage [10], Microsoft Azure Storage [14], Rackspace Cloud Files [21], and Softlayer Cloud Storage [22]. ICS was configured to use the first three of them ($n = 3$); the Two-Step Full Replication (2S-FR), Three-Step Erasure Codes (3S-EC) and DepSky protocols used the first four clouds mentioned ($n = 4$); and the Two-Step Erasure Codes (2S-EC) protocol used all of them ($n = 5$).

8.6.2 List Quorum Performance

One of the main differences between our protocols and the other MW-regular register of the literature, namely ICS [54], is the fact that garbage collection is decoupled from write operations. This means that **list** operations invoked in ICS' base objects always return a small number of keys. Since in our protocols the garbage collection is executed in background, it is important to understand how the presence of obsolete keys (not garbage collected) in the KVSs affects the performance of list the available keys.Notice this issue does not affect DepSky as it does not use the **list** operation [65].
Figure 8.2 shows the latency of executing the *listQuorum* function with different numbers of keys stored in the KVSs, for our three protocols (which consider different quorum sizes). As can be seen, 2S-EC presents the worst performance, indicating that listing bigger quorums is more costly. We can also observe that the performance degradation of the **list** operation when there are less than 100 obsolete versions is very small (specially for 2S-FR and 3S-EC). However, the latency is roughly 2× and 4× worst for 500 and 1000 versions, respectively. This is an indication that triggering the garbage collection once on every 100 write operations will avoid any significant performance degradation.

8.6.3 Read and Write Latency

Figure 8.3 shows the write and read latency of our three protocols, ICS [54] and DepSky [65], considering different sizes of the stored data.
The results show that, when reading 64kB and 1MB, 2S-FR and 3S-EC presents almost the same performance, while 2S-EC is slightly slower, due to the use of larger quorums. This means that reading only one data value with a full copy of the data is as fast as reading $f + 1$ blocks with half of



(a) 64kB. (b) 1MB. (c) 16MB.

Figure 8.3: Median and 90-percentile latencies for read and write operations of register emulations.

the size of the original data. This is not the case for 16MB data. The results show it is faster to read $f + 1$ data blocks of 8MB in parallel from different clouds (2S-EC and 3S-EC) than reading a 16MB object from one cloud (2S-FR).

For writing 64kB objects 3S-EC is slower than 2S-FR and 2S-EC. This happens due to the latency of the third step of the protocol (write of the proof object). When writing 1MB objects, our three protocols present roughly the same latency, being the 3S-EC protocol a little bit slower for the median-percentile (also due to the write of the proof object). However, when clients write 16MB data objects, the additional latency associated with this third step is negligible. Overall, these results can be explained by the fact that the proof object has zero bytes. Thus, 3S-EC protocol presents the best performance due to its use of dissemination quorums and erasure codes. For this data size, the 2S-FR protocol presents the worst performance of our protocols as it stores a full copy of the data in all clouds.

The key takeaway here is that our protocols present a performance comparable with DepSky [65] (Dep), which does not support multiple writers, and a performance up to 2× better than the crash fault-tolerant MW register presented in [54] (ICS). On the other hand, ICS presents the worst latency among the evaluated protocols. One of the main reasons for this to happen is the fact that it does not use erasure codes. Furthermore, for reading, this protocol always waits for a majority of data responses, which makes it slower than, for example, the 2S-FR that only waits for one valid **get** response. In turn, for writing, ICS writes the full copy of the data twice on each KVS to deal with the garbage collection racing problem, removing also obsolete versions.

8.6.4 Read Under Write Contention

Figure 8.4 depicts the read latency of 1 MB objects in presence of multiple contending writers. This experiment does not consider DepSky as it only offers SW semantics.

The results show that both 2S-FR and 2S-EC read latencies are affected by the number of contending writers. This happens for two reasons: (1) under concurrent writes, these read protocols commonly try to first read incomplete versions from the KVSs before finding a complete one (i.e., the loop on read protocols is executed more than once); (2) since we are not garbage collecting obsolete versions, more writers send more versions to the clouds, negatively influencing the *listQuorum* function latency. Since 3S-EC is not affected by the first factor, its read operation performs slightly better with contending writers.

ICS's read presents a constant performance with the increase of contending writers, however, 2S-FR and 2S-EC present competitive results and 3S-EC presents results always better than it, even without garbage collecting obsolete versions.

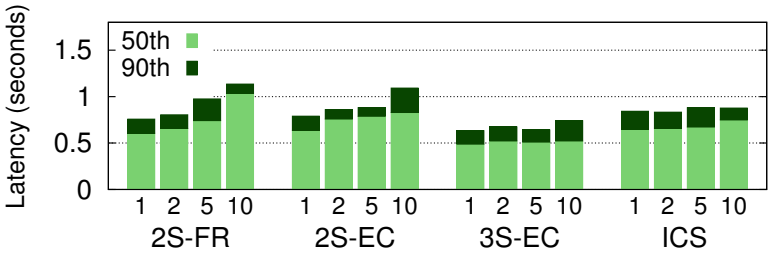


Figure 8.4: Median and 90-percentile read latencies in presence of contending writers.

8.7 Conclusion

This chapter presented a study of fundamental storage abstractions resilient to Byzantine faults in the data-centric model, with applications to cloud-of-clouds storage. In this context, we presented three new register emulations: (1) one that uses dissemination quorums and replicates full copies of the data across the clouds, (2) another that uses masking quorums and reduces the space complexity through the use of erasure codes, and (3) a third one that increases the number of accesses made to the clouds to use dissemination quorums together with erasure codes.
Our evaluation shows that the new protocols have similar (or even better) performance and storage requirements than existing emulations that either support a single writer [54] or tolerate only crashes [65].

Chapter 9 Low-cost Cloud-based Disaster Recovery for Databases

9.1 Introduction

The occurrence of disasters introduces some serious challenges to the design of IT systems. In opposition to other sources of failures, disasters affect the whole (or at least a big part of the) infrastructure where the system is hosted, resulting in greater damage to the service provided. Consequently, the ability to tolerate disasters requires specific data protection mechanisms and careful planning [194].
More specifically, tolerating disasters requires placing (backup) resources in a geographically separated location so that the same disaster does not affect the primary and the backup infrastructures. Such approach results in significant additional costs, and thus it is not used by budget-constrained services. The emergence of cloud computing made it possible to implement disaster recovery (DR) with a small fraction of the costs of a dedicated infrastructure [316]. System operators can thus rely on cloud providers to host a portion (or even full copies) of their system and, if the primary site goes offline, they can quickly assume the service provision.
Cloud-based disaster recovery mechanisms require different approaches to deal with stateless and stateful services. For the former, administrators only have to store server VM images to enable the services to be started when required. For stateful services, there are basically two options: periodically storing state snapshots, or maintaining a warm backup on the cloud. The first approach is know as backup and restore while the later is sometimes called Pilot light, in the sense that this backup can spark a whole backup infrastructure if needed [271]. The replication protocol for maintaining such replica in the cloud can be implemented at different layers, such as within the service itself [20, 18, 195], in the virtualization platforms [267, 317] or in the storage level [256, 186].
Despite all these options, data loss is still a common event that may lead to severe consequences. Although statistics about data losses and its effects are sometimes misleading [149], recent surveys showed that data loss costs $1.7 Trillion per year for medium and big companies [199]. A few years ago a survey by Symantec showed that 40% of Small and Medium Enterprises (SMEs) do not do regular backups [294]. We believe the situation improved in the last years, but it is unlikely that this protection gap disappeared. A more recent survey revealed that 58% of the SMEs could not sustain any amount of data loss [176], and that 62% of these companies do not backup their data on a daily basis. These numbers clearly indicate that even simple backup routines are still a challenge for SMEs, and reveal that fully automated disaster recovery solutions are not yet widely deployed. This landscape is even worse if one considers new data loss threats, such as ransomware, which is becoming a plague for SMEs [295]. Lack of budget and automation are usually pointed as key challenges for implementing effective business continuity plans [28].
Within the work described in this chapter we try to improve this situation, specially for SMEs and other organizations that cannot afford the cost and complexity of geo-replication and existing DR solutions, through the exploitation of two facts. First, non-VM-based cloud services have the potential of reducing drastically the costs of a disaster recovery solution. Second, most business critical data are stored in database management systems (DBMS), therefore, it is paramount for any serious DR solution to protect these systems.
In this context, we present Ginja, a disaster recovery system for transactional DBMSs based on

popular cloud object storage services such as Amazon S3, Azure Blob Storage or Google Storage. GINJA was designed with four objectives in mind: low operational costs, fine-grained control over the data that can be lost due to a disaster, low performance overhead, and high portability among different DBMSs.
GINJA exploits a new region in the design space of disaster tolerance/recovery systems, right between backup and restore and pilot light solutions. In particular, it has costs close to the former (i.e., maintaining a backup database snapshots in the cloud) with the same control over data loss and performance of the later (i.e., having a passive database replica in a VM in the cloud). To design such system in a portable way, we had to overcome several challenges: (1) define means to capture all the relevant I/O from the DBMS, (2) map these updates to a data model implementable in the clouds' object storage interface, and (3) provide algorithms and mechanisms for controlling the behavior of the system to match different requirements and budgets. In the end, our design exposes the cost vs. performance vs. maximum data loss tradeoff in two parameters, allowing a tight control of these factors.
We have implemented and evaluated a prototype of GINJA supporting PostgreSQL [19] and MySQL [17] to show that our solution is feasible, performant and cost-efficient. For instance, we are able to provide DR for many database setups relevant to SMEs (e.g., databases with up to 30GB of size and hundreds of updates/minute) costing only one dollar/month, which is 14× less than the cost of running the cheapest EC2 VM for a month. Furthermore, our results show that using GINJA leads to a small performance loss for the TPC-C benchmark and minor additional CPU and memory usage on the database server. Although our present implementation only supports PostgreSQL and MySQL, it was designed to be easily extended for other DBMS.

9.2 Disaster Recovery

A Disaster is an event that has a negative impact on organizations' business continuity and/or finances [271]. Examples of disasters include network and power outages, hurricanes, earthquakes, floods, and so forth. Disaster Recovery (DR) is the area that makes IT systems tolerant and recoverable from the damages caused by these disasters. This is mainly achieved by having a Primary Site infrastructure to respond in normal operation, plus a Secondary Site (or Backup Site) in a geographically-distant location [99, 194].
Yet, different systems have different disaster recovery requirements [256]. Such requirements include recovery time, consistency degree of the data recovered, performance impact during normal operation, distance between sites and costs. For defining such requirements there exist two main parameters [99]: Recovery Point Objective (RPO), which is the amount of updates (measured in time) that can be lost due to a disaster; and Recovery Time Objective (RTO), which refers to the duration of downtime that is acceptable before a system recovers from a disaster.
There are several ways to implement a disaster recovery strategy [194]. The classical approach is the Tape Backup and Restore [286]. This technique consists of periodically taking consistent snapshots of the data (optionally interspersed with incremental backups), storing it into tape drives and sending those tapes off site. Although this approach is attractive for being low-cost, it has the disadvantages of having long recovery time and always restoring the system to an outdated state, as backup intervals are typically long. An alternative strategy is Remote Mirroring [186]. In this approach, the system continuously replicates its data to an online remote mirror, which ensures the continuity of the system if a disaster occurs. Despite being usualy more expensive, this technique can substantially reduce both the RPO and RTO when compared with tape backup and restore.
The data replication between sites can be performed essentially in two ways: synchronously or asynchronously [317, 76] (also called Eager and Lazy replication in the database community [195]). In Synchronous Replication, the system loses performance as the primary site can only return successfully from a write operation after it has been acknowledged by the secondary site. In Asynchronous Replication the primary site is allowed to proceed its execution without waiting for the replication to

be completed at the secondary site. This type of replication overcomes the performance limitations of synchronous replication at the expense of allowing recent updates to be lost if a failure occurs.
Public clouds appear as a perfect solution for implementing DR mechanisms. The main reasons are their large portfolio of services (e.g., object storage, computing, networking, database, queue services), relatively user-friendliness, security, multi-site availability, and the pay-as-you-go cost model. These factors allow the design of DR solutions suitable for each organization regarding its objective (RPO and RTO) and budget [271]. The simplest (and probably the cheapest) possible example is the storage of data backups in cloud storage services such as Amazon S3. A more evolved (and expensive) solution considers a subset of services replicated in VMs running in the cloud (e.g., on Amazon EC2). This latter technique allows a low data loss and recovery time in case of a disaster.
Some public clouds also provide disaster recovery services that typically use their infrastructures as a secondary site. Examples of such services are Azure Site Recovery [14] and vCloud Air Disaster Recovery [24]. These services automate the configuration and management of the cloud backup infrastructure, but their cost is equivalent or even higher than maintaining plain backup VMs in the cloud.
It is also possible to run the primary site of a system entirely in a cloud. However, this approach does not eliminate the need for disaster recovery since cloud wide outages, although rare, are a potential threat to systems that rely entirely on one cloud infrastructure to perform its functions [65].
The more cloud resources a system requires in failure-free operation, the higher will be the costs of the DR solutions. Even if the costs during failover are slightly higher in cloud based solutions, the overall costs can still be smaller since disasters are supposed to be rare events [316].

9.3 Low-cost Cloud-based Disaster Recovery

In this work, we advocate that current cloud-based disaster recovery solutions are much more expensive and difficult to manage than they should be. In particular, we show that a full-fledge database disaster recovery system could be implemented without requiring any dedicated VM in the cloud. Such system works as follows. Initially, a copy of all database files is uploaded to a cloud storage service (e.g., Amazon S3), and then, as updates are committed to the log file, the system sends them to the cloud as commit objects. As new updates keep being performed, the DBMS executes a checkpoint to update the table files and to clean its commit log. In this situation, the system updates the database files in the cloud and removes outdated commit objects. In case of disaster, the recovered database needs to be able to figure out the pre-disaster state using the objects stored in the cloud.
Such system could be extremely cheap if one considers that some recent updates could be lost in case of a disaster (as in most DR systems). This would enable the DR system to send batches of updates to the cloud periodically.
As an illustrative example, consider that someone wants to spend a maximum of **$1 per month** in a database DR solution. In October 2016, Amazon S3 standard storage costs are $0.03 per GB/month, $0.01 per 1000 file uploads, and free upload bandwidth and delete operations [3].[1] Considering this, it is possible to plot the capacity of a database (in terms of size and number of cloud synchronizations per hour) for such one-dollar budget, as shown in Figure 9.1.
In the figure, every point below the line represents a setup costing less than $1 per month. For example, this budget is enough to protect a database with 4.5GB with two synchronizations per minute (setup C), or a 19GB database with one synchronization per minute (setup B), or even a 28.5GB database synchronized once every three minutes (setup A). Importantly, an organization whose activity happens mostly from 9AM to 5PM (which is the case for many non-online business) can have roughly three more synchronizations per hour during these hours.
Notice that these setups still provide acceptable RPOs for many SMEs and other medium-size organizations. In any case, by understanding what one can have with $1 per month, it is possible to assess the cost of more demanding setups (i.e., larger databases or smaller RPOs).

[1] Other services such as Azure Storage, Google Storage and Rackspace Files offer similar price models. GINJA can be used with any of them.

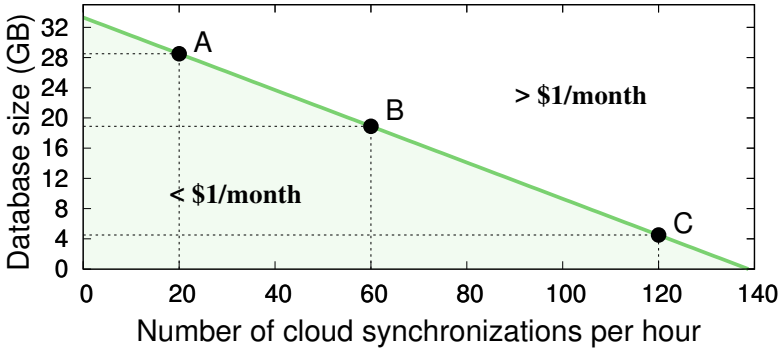


Figure 9.1: Database size and number of cloud synchronizations per hour in a S3-based DR solution with a $1 monthly budget.

The system proposed in this chapter, Ginja, exploits this low-cost opportunity to enable any small and medium organization running a relational DBMS to have a DR solution almost for free, and with close-to-zero management effort.

9.4 Transactional Database I/O

Ginja is a DR system designed for database management systems. The integration between our system and a DBMS happens at the file system level. This allow us to intercept every file system call performed by the DBMS on the database-related files. In this section, we describe the kind of DBMSs our system assumes and discuss how PostgreSQL and MySQL fits in this model.

We consider transactional databases that implement data durability using a set of table files and a Write-Ahead Log (WAL) divided in several segment files [239, 114]. The I/O on these files is performed on the granularity of a page, which is composed by many records. Every time a transaction is committed, the only important I/O performed is a synchronous write to a WAL file segment. All the table pages remain in memory until a periodic checkpoint occurs. When this happens, the pages are written to the table files, and a special record is inserted in the WAL marking that everything before this record is already in durable memory.

Implementing a DR solution with fine-grained control of the database RPO without changing the DBMS requires a deep understanding on how databases access these files. More precisely, there are at least three types of events that need to be detected. The first one is an update commit, when a record is written to the WAL. The second one corresponds to the write that marks the start of a checkpoint. The last type of event we need to detect is the last write of a checkpoint, i.e., the last write after which it is safe to delete old WAL entries. Table 9.1 describes these events for the databases we use in this work: PostgreSQL and MySQL.

PostgreSQL [290, 20] keeps its log segments in a set of `x_log` files (pages of 8kB), and periodically (with a configurable period), writes the dirty pages (also 8kB) to the table files. Additionally, it uses a `pg_log` file to store the status of each transaction (the checkpoint starts with a write in this file) and a small `pg_control` file to store a pointer to the last checkpoint record in the WAL, marking the starting point on the WAL upon a recovery. A write to `pg_control` marks the end of a checkpoint.

MySQL supports different types of storage engines, being InnoDB the standard one for having ACID transactions [17]. MySQL/InnoDB (or simply MySQL) writes all committed transactions to an `ib_logfile` file (in pages of 512 bytes), and executes checkpoints quite differently from PostgreSQL. More specifically, the system can flush modified database pages (of 16kB) to their respective files at any moment, in small batches. This mechanism is known as fuzzy checkpoint [16]. The fact checkpoints are "opportunistic" makes their write pattern a bit more complicated and variable than the ones in PostgreSQL. However, as can be seen in the Table 9.1, it is possible to detect their start and finish by verifying a handful of conditions.

Event	PostgreSQL	MySQL
Update commit	sync. write to a `pg_xlog` file	sync. write in one of the `ib_logfile` files (except the header of the `ib_logfile0`)
Checkpoint start	sync. write to a `pg_clog` file	sync. write to one of the data files (`ibdata`, `.ibd`, and `.frm`)
Checkpoint end	sync. write to the `global/pg_control` file	sync. write in the offset 512 and/or 1536 of the `ib_logfile0` file

Table 9.1: How GINJA detects the three most important DBMS events in PostgreSQL and MySQL.

9.5 Ginja

GINJA intercepts the I/O performed by the DBMS and backs up the relevant data to a cloud storage service in a cost-efficient manner. Although our implementation consists in an application-specific FUSE file system (see §9.6 for details) able to capture the semantics of the database's I/O operations without having to change the DBMS, our design is generic and only assumes that the events of Table 9.1 are intercepted.
GINJA relies on cloud storage services (e.g., Amazon S3, Azure Blob Storage) to store its data in a remote site. As described before, we choose such services as secondary infrastructure because they have the potential of lowering both the monetary and management costs of our DR solution.
This decision fundamentally differs GINJA from existing works on cloud disaster recovery [256, 186, 317, 237], and impacts our design in three important ways. First, storage clouds provide REST interfaces containing only a few basic operations (PUT, GET, LIST, and DELETE). Consequently, we have to implement all DR control at the primary side (i.e., at the client side). Second, we must make as few assumptions as possible about the underlying storage clouds, so that our clients can choose the cloud provider they want with few or no modifications to our code. Finally, it is crucial that we take into account the pricing model of the cloud storage services when performing cloud operations, to reduce costs as much as possible.

9.5.1 Controlling Costs and Data Losses

GINJA deals with the fundamental trade-off between performance and data protection by allowing users to decide the maximum amount of recent updates that can be lost when a disaster occurs. Thus, instead of following a completely synchronous or asynchronous approach, we defined a model that allows users to choose the desired synchronization level. Furthermore, as sending data to the cloud has its costs, our model also delegates to users the performance-cost trade-off. This model includes two parameters:

- Batch—the maximum number of database updates included in each cloud synchronization;
- Safety—the maximum number of database updates that can be lost in the event of a disaster.

Batch defines how often WAL writes are sent to the cloud, whereas Safety defines the durability guarantees provided by GINJA. These parameters define a threshold of database updates that trigger GINJA to perform its actions, even when the DBMS receives bursts of write requests.
Batch and Safety can be defined both in terms of number of updates – B and S – and time – T_B and T_S – working as follows. A batch of updates is sent to the cloud if B updates are executed or if there are some updates to be sent to the cloud and T_B seconds have elapsed since the last synchronization ended. Similarly, a WAL write performed by the database blocks if there are more than S updates that are not confirmed to have been written to the cloud or if there are some updates to be sent to the cloud and T_S seconds have elapsed since the first non-synchronized update was executed. Notice that in intensive workloads, only B and S will be relevant as there will be no time for timeouts.
Figure 9.2 illustrates how these parameters work. Whenever B operations are executed in the DMBS, GINJA performs a cloud synchronization and allows the database management system to proceed its normal operation. On the other hand, when the S^{th} database update since the last successful synchronization is submitted (U_{21} in the figure), our system blocks the DBMS until a positive acknowledgment is received from the pending cloud synchronizations.

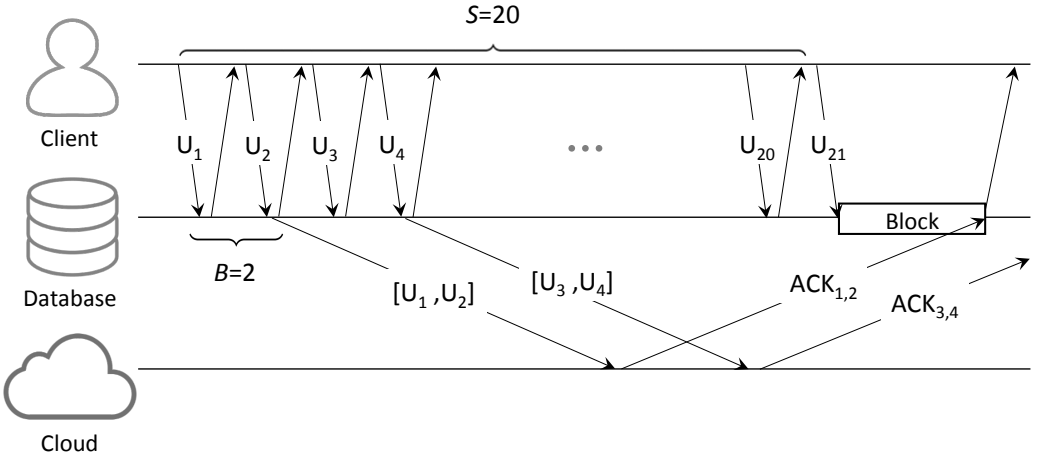


Figure 9.2: Influence of B (Batch) and S (Safety) in the execution of Ginja. In this example $B = 2$, thus each cloud backup includes two database updates. It is possible to observe that Ginja blocks the DBMS whenever more than $S = 20$ database updates are executed without being acknowledged by the cloud.

Ideally, B should be substantially lower than S so that Ginja does not block or interfere with the DBMS performance during regular operation.

9.5.2 Data Model

Ginja uses a specialized data model that allows the synchronization of file updates as they are issued locally, and the reconstruction of those files from the objects present in the cloud when necessary. This model aims to reduce the total volume of data kept in the cloud, and to minimize the number of cloud operations executed (as these are the only factors that influence Ginja's monetary cost in the absence of disasters). The data model considers two types of objects:

- WAL Objects—contain data written to the local WAL segments. The content of each local WAL segment is stored in several WAL objects (one for up to B updates). The WAL objects are named following the format `WAL/<ts>_<filename>_<offset>`, in which the `ts` establishes total order on the WAL objects, `filename` is the name of the corresponding WAL segment, and `offset` is the position of its content in the segment.
- DB Objects—store information relative to all relevant database files excluding the WAL segments. There are two types of DB objects: dumps and incremental checkpoints. The DB objects are named following the format `DB/<ts>_<type>_<size>`, containing thus its `ts`, `type` ("dump" or "checkpoint"), and `size`. In this case, the `ts` corresponds to the timestamp of the last uploaded WAL object before the beginning of the checkpoint.

We limit the maximum size of each cloud object to a configurable limit to optimize the upload time [177], and consequently, the Ginja performance.[2]

9.5.3 Algorithms

This section details Ginja algorithms for initialization and recovery, update processing and checkpoint management.

9.5.3.1 Initialization.

Algorithm 5 describes how Ginja is initialized in its different modes (Boot, Reboot and Recovery).

[2]We hide this aspect from ours algorithms (see §9.5.3) for sake of simplicity.

When started, and before engaging in one of its three initialization modes, the system first initializes the cloudView data structure as empty. This structure is used to keep track of the existing WAL and DB objects in the cloud, and is updated every time a cloud operation is performed. After that, all threads required in the system are started (Lines 2–6).
The Boot mode is used to create a dump of an existing database on the cloud. Concretely, the system creates a set of WAL objects (one for each local WAL segment), and one dump DB object (Lines 7–18). Only after all the objects are successfully uploaded to the cloud the file system is mounted and the DBMS can be started.
The Reboot mode can be used to restart the system after a safe stop of the DBMS. This mode assumes that the data on the cloud is synchronized with the local files of the database. Therefore, the only required step is to update the cloudView by listing the objects present in the cloud (Lines 19–22).
The Recovery mode is used to rebuild the database files from the objects stored in the cloud. The first step is to list the objects in the cloud and update the cloudView data structure (Lines 24–26). Then, the database files are reconstructed from the most recent dump in the cloud (Lines 27–29) and, afterwards, these files are updated with the incremental checkpoint objects (Lines 30–36). Finally, GINJA downloads the WAL data objects written after the last checkpoint and rebuild the local WAL segments following the ts ordering so that the DBMS can perform crash recovery (Lines 37–40).

9.5.3.2 Database Update Commits.

Algorithm 6 describes how GINJA processes the intercepted writes to WAL segment files without violating the parameters B, S, T_B and T_S.
When the system intercepts an update to the WAL segment file it writes the data on the local copy of the file and enqueues the update to be sent to the cloud (Lines 4–6). The operation only returns to the DBMS if the S and T_S parameters are not violated, otherwise the systems blocks (Line 7) until the pending writes are successfully uploaded.
Lines 8–22 show how the commits are processed. First, the writes are aggregated respecting B and T_B (Lines 9–12). Notice that this aggregation may result in more than one object because the batch of committed updates could belong to different WAL segments.[3] After being aggregated, they are sent to the cloud (Lines 13–16).
The aggregation is important because the DBMSs write to the log on the granularity of a page, and many times these pages are overwritten with more updates. Consequently, by aggregating them we coalesce many updates in a single cloud object upload. This reduces the storage used and the total number of PUTs executed in the cloud and, consequently, the monetary cost of our DR solution.
Additionally, it is possible to have several threads uploading cloud objects in parallel (Lines 4–5 of Algorithm 5), which brings great benefits in terms of performance [177]. However, it is no longer guaranteed that the WAL objects are uploaded following the timestamp order (i.e., the *ts* obtained on Line 14). In the worst case scenario, a disaster may occur in the moment when the most recent WAL updates are already replicated in the cloud, while others with smaller timestamps are still in transmission. During Recovery, GINJA deals with this incomplete state by downloading only the WAL objects that have consecutive timestamps. Consequently, to guarantee that the maximum amount of updates lost in case of disaster do not exceed Safety, GINJA unblocks the DBMS only after uploading all WAL objects with consecutive ts values. This can be observed in Algorithm 6: the variables that control these parameters (specifically `commitQueue.size`, $\texttt{timeoutT}_\texttt{S}$ and the timer of $\texttt{TaskT}_\texttt{S}$) are reset (unblocking the DBMS) if all WAL objects previously uploaded can be used to recover from a disaster that would occur immediately (Lines 20–22).

[3] WAL segments are typically much larger (e.g., 16MB in PostgreSQL and 48MB in MySQL by default) than the page size. Consequently, this aggregation results normally in only one cloud object.

Algorithm 5: Initialization tasks.

```
1  cloudView ← ∅;                                          // Used in all Algs.
2  TaskT_B.startTimer(T_B);                                // Used in Alg. 6
3  TaskT_S.startTimer(T_S);                                // Used in Alg. 6
4  for 1 ≤ i ≤ nThreads do
5      CommitThread_i.start;                               // Used in Alg. 6
6  CheckpointThread.start;                                 // Used in Alg. 7
7  Mode Boot begin
8      currentTs ← 0;
9      for each file in Local WAL Segments, in increasing order do
10         objName ← "WAL/"+currentTs+"_"+file.name+"_0";
11         cloud.PUT(objName, file.content);
12         cloudView.addWAL(currentTs, file.name, 0);
13         currentTs ← currentTs + 1;
14     dbObject ← ∅;
15     for each file in Local DB Files do
16         dbObject.add(file.name, file.content);
17     cloud.PUT("DB/0_dump_"+dbObject.size, dbObject);
18     cloudView.addDB(0, "dump", dbObject.size);
19 Mode Reboot begin
20     cloudList ← cloud.LIST();
21     for each obj in cloudList do
22         cloudView.add(obj);
23 Mode Recovery begin
24     cloudList ← cloud.LIST();
25     for each obj in cloudList do
26         cloudView.add(obj);
27     dump ← cloud.GET(mostRecentDump(cloudList.dbObjects));
28     for each file in dump do
29         writeLocally(file.name, 0, file.content);
30     checkpoints ← newerThan(cloudList.dbObjects, dump.ts);
31     maxCkptTs ← dump.ts;
32     for each obj in checkpoints, in increasing ts order do
33         currentCkpt ← cloud.GET(obj);
34         for each file in currentCkpt do
35             writeLocally(file.name, file.offset, file.content);
36         maxCkptTs ← obj.ts;
37     segments ← newerThan(cloudList.walObjects, maxCkptTs);
38     for each obj in segments, in increasing ts order and with no gaps do
39         content ← cloud.GET(obj);
40         writeLocally(obj.filename, obj.offset, obj.content);
```

9.5.3.3 Checkpoints and Garbage Collection.

Algorithm 7 describes how GINJA handles checkpoints. As performance is one of our key concerns, we decouple as much as possible the (local) DBMS checkpoints from the writing of checkpoints to the cloud (Lines 6, 20–21). Therefore, checkpoint data is collected as the files are updated during a checkpoint (Lines 3–16) and, when the checkpoint is finished locally, a separate thread is used for sending the updates to the cloud as DB objects (Lines 17–29). Notice the checkpoint start and end conditions (see Table 9.1) are verified in Lines 4 and 8.

There are two ways of sending checkpoint data to the cloud: as a checkpoint or as a dump. Whenever the total size of the DB objects in the cloud is greater or equal to 150% of the local database size, GINJA creates a new database dump (Lines 9–11). Otherwise, it creates an incremental checkpoint. In the first situation, GINJA will not execute any write in the local DB files while the dump object is being created, to guarantee that the database is dumped in a consistent way. This does not block database commits as WAL file writes are mostly independent of checkpoint processing (at least in the two databases we support).

Every time a DB object with timestamp *ts* is completely uploaded to the cloud, GINJA removes all WAL objects with timestamps up to *ts* (Lines 4–5 and 23–25). This is safe because such WAL objects

Algorithm 6: Database Commits.

```
1  commitQueue ← ∅;     // Holds all the pending synchronizations
2  timeoutT_S ← false;
3  timeoutT_B ← false;
4  When write(WAL_segment, offset, content) is intercepted begin
5      writeLocally(WAL_segment, offset, content);
6      commitQueue.put(⟨WAL_segment, offset, content⟩);
7      wait until commitQueue.size ≤ S and timeoutT_S = false;
8  CommitThread Execution begin
9      Loop
10         wait until commitQueue.size ≥ B or timeoutT_B = true;
11         updates ← commitQueue.getNextBatch();
12         aggUpdates ← aggregateUpdates(updates);
13         for each u in aggUpdates do
14             ts ← cloudView.getNextWALts();
15             cloud.PUT("WAL/"+ts+"_"+u.filename+"_"+u.offset);
16             cloudView.addWAL(ts, u.filename, u.offset);
17         TaskT_B.resetTimer();
18         timeoutT_B ← false;
19         wait until commitQueue.lastBatchElements() = updates;
20         commitQueue.removeLastNElements(updates.size);
21         TaskT_S.resetTimer();
22         timeoutT_S ← false;
23 TaskT_B (upon timeout) begin
24     if commitQueue.size > 0 then
25         timeoutT_B ← true; // Trigger an upload
26 TaskT_S (upon timeout) begin
27     if commitQueue.size > 0 then
28         timeoutT_S ← true; // Block the DBMS
```

Algorithm 7: Checkpoints and Garbage Collection.

```
1  checkpointQueue ← ∅;
2  timestamp ← ∅;
3  When write(dbFile, offset, content) is intercepted begin
4      if ⟨dbFile, offset, content⟩ is the first write in checkpoint then
5          timestamp ← cloudView.getLastWALts();
6      writeLocally(dbFile, offset, content);
7      dbObject ← addAndAggregate(⟨dbFile, offset, content⟩);
8      if ⟨dbFile, offset, content⟩ is the last write in checkpoint then
9          if cloudView.getTotalDBSize() ≥ 150%× local DB size then
10             dbObject ← create dump from local DB files;
11             dbObject.type ← "dump";
12         else
13             dbObject.type ← "checkpoint";
14         dbObject.ts ← timestamp;
15         checkpointQueue.add(dbObject);
16         dbObject ← ∅;
17 CheckpointThread Execution begin
18     Loop
19         wait until checkpointQueue.size > 0;
20         obj ← checkpointQueue.remove();
21         cloud.PUT("DB/"+obj.ts+"_"+obj.type+"_"+obj.size, obj);
22         cloudView.addDB(obj.ts, obj.type, obj.size);
23         for each walObject ∈ cloudView : walObject.ts ≤ obj.ts do
24             cloud.DELETE(walObject.objectName);
25             cloudView.delete(walObject);
26         if obj.type ="dump" then
27             for each dbObject ∈ cloudView : dbObject.ts < obj.ts do
28                 cloud.DELETE(dbObject.objectName);
29                 cloudView.delete(dbObject);
```

contain information that will not be used in a recovery. Additionally, when the uploaded DB object is a dump, all the previous DB objects (incremental checkpoints and the previous dump) are deleted as well (Lines 26–29).

9.5.4 Extensions

In the following we describe some extensions to the basic disaster recovery algorithms described in previous section.

9.5.4.1 Compression and encryption.

Ginja supports the compression and/or encryption of WAL and DB objects before their write to the cloud. Compression decreases the data size and is straightforward to implement. Encryption, on the other hand, requires the management of a local secret key that cannot be stored in the cloud to preserve the database confidentiality. Ginja uses a key generated from a password (assumed to be kept secure) provided during the initialization of the system. At runtime, this key is kept in memory and never written to any local or remote file.
Our system also implements some basic integrity protection by storing a MAC of each object together with it. If encryption is enabled, the provided password is also used to generate the MAC key, otherwise, a default string (a configuration parameter) is used to generate this key.

9.5.4.2 Point-in-time recovery.

The garbage collection algorithm discussed in the previous section deletes all outdated objects when a new checkpoint is written to the cloud. However, the algorithm can be easily modified to delete only certain objects and keep others to allow the recovery of the system to a certain point in time. More specifically, Lines 23–29 of Algorithm 7 can be modified to keep the database state on date-time T by finding the first object o stored in the cloud after T and keeping (1) the most recent dump d written before this object, (2) all incremental checkpoints written between d and o, and (3) all WAL objects written between the last incremental checkpoint and o.
As expected, storing snapshots for point-in-time recovery might substantially increase the cloud storage costs, unless they are moved to cold storage (e.g., Amazon Glacier).

9.5.4.3 Backup verification.

One of the key concerns in every disaster recovery plan is how to ensure the plan will work when a disaster strikes. An important feature of Ginja is that it allows the verification of a database backup in an easy and cheap way, without interfering with the production system.
To do that we just need to start a replica of the database in recovery mode and run a set of service-specific tests. This implies in a sequence of three validations:

1. Every object downloaded from the cloud has its integrity validated through its MAC verification;
2. The DBMS itself verifies the integrity of the tables and WAL segments when restarting the system with the local files rebuilded by Ginja;
3. Once the DBMS starts, a pre-prepared script can run a series of queries to assess if recent updates are available on the database. This verification can be made automatically using some service-specific heuristic and the result of the script can be sent to an administrator for verification.

The cost of database verification is basically the cost of downloading the database objects to a local machine or the cost of running a VM in the same cloud (if appropriate). In any case, the verification procedure can be fully automated.

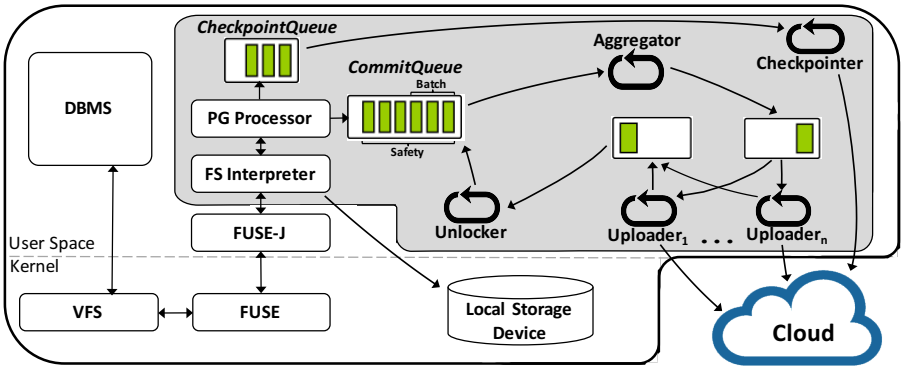


Figure 9.3: Detailed architecture of Ginja.

9.6 Implementation

Ginja was implemented as a File System in User Space [299] using approximately 4000 lines of Java code distributed in 32 files. Most of this code is DBMS-agnostic and there are only two small modules that are specific for processing I/O from PostgreSQL and MySQL/InnoDB, with around 200 lines of code each. The cloud synchronization module is based on an external library to execute cloud operations (we use DepSky's cloud storage drivers [65]). Furthermore, our current prototype implements compression using ZLIB configured for fastest operation, encryption using AES with 128-bit keys, and MACs using SHA-1.

Figure 9.3 presents the internal architecture of Ginja. The FS Interpreter implements a FUSE-J interface [9], and is responsible for three main tasks: (1) intercepting the file system calls performed by the DBMS; (2) accessing the local disk; and (3) forward a well-formatted data do the database processor. In this way, Ginja can be easily extended to support other DBMS by implementing new processors.

The implementation of a processor is a relatively simple and straightforward procedure. However, this requires an in-depth knowledge of the DBMS I/O management. The processor uses two different queues to put the data received from the file system: one for the WAL writes and another for the checkpoint writes.

The write operations performed in the WAL are sent to a queue named CommitQueue. This data structure has a maximum capacity of S elements, and only supports getting B elements at a time. Any attempt to put an element into a full CommitQueue will block. Likewise, attempts to take elements from a CommitQueue with less than B elements will result in blocking until the T_B' timer expires.

A thread called Aggregator is responsible to read sets of B updates from this queue (without removing them), aggregate those writes into a single object, and submit the resulting data to a second queue. A number of Uploader threads will retrieve elements from this queue and upload them in parallel as WAL objects, submitting an acknowledgment to a third queue whenever a cloud upload completes. Last but not least, a thread called Unlocker will remove sets of Batch elements from the head of CommitQueue, according to the acknowledgments received by the Uploader threads. In the end, the Aggregator, Uploader and Unlocker threads implement Algorithm 6.

The write operations performed during checkpoints are enqueued to the CheckpointQueue so that a thread called Checkpointer aggregates the data and uploads it to the cloud in the form of DB cloud objects (this thread implements Algorithm 7).

9.7 Cost Analysis

A key objective of our work is to reduce the operational cost of the database disaster recovery solution. In this section, we consider the operational cost of the system.

9.7.1 Ginja Cost Model

The factors that influence the operational cost of Ginja in the absence of disasters are the storage used to keep WAL and DB objects in the cloud, as well as the amount of PUT operations used to upload the WAL and DB data. Thus, the monthly operational cost of our system is given by the following equation:

$$C_{Total} = C_{DB\_Storage} + C_{DB\_PUT} + C_{WAL\_Storage} + C_{WAL\_PUT}$$

Let us now explore in detail how each of the four factors of this equation can be calculated.

9.7.1.1 Storage of DB objects.

Ginja uploads the information of the database files in the form of DB objects. The cost of storing these objects is given by the following equation:

$$C_{DB\_Storage} = \frac{DBSize \times 1.25}{CR} \times C_{Storage}$$

The DB_Size is measured in GBs and the $C_{Storage}$ in \$/GB/month. The main factor that influences this cost is the size of the database. Recall that Ginja ensures that the maximum volume that the DB objects can take in the cloud is 150% of the local database size (due to the incremental checkpoints). As a result, the average DB storage in the cloud will be 25% greater than the database size. Additionally, the DB data size can be further reduced by using compression (the compression rate, CR in the equation).

9.7.1.2 PUT operations of DB objects.

The number of PUT operations used to upload DB objects depends essentially on how often checkpoints occur, the average checkpoint size, and the price of each PUT operation. The cost of this component can be calculated as follows:

$$C_{DB\_PUT} = \frac{30 \times 24 \times 60}{Ckpt_{Period}} \times \left\lceil \frac{CkptSize}{20MB} \right\rceil \times C_{PUT}$$

The first fraction of this equation gives us the number of checkpoints that the DBMS performs per month (note that $Ckpt_{Period}$ is given in minutes). The second fraction determines the number of PUT operations executed in each checkpoint, i.e., number of uploaded DB objects split in files of up to 20MB.

9.7.1.3 Storage of WAL objects.

The third cost factor of Ginja is the volume of the WAL objects present in the cloud, calculated as follows:

$$C_{WAL\_Storage} = \left(\left\lceil \frac{W \times Ckpt_{Time}}{RecPerPage} \right\rceil + 1 \right) \times \frac{PageSize}{CR} \times C_{Storage}$$

The first part of the equation determines the maximum number of WAL segments that can be in the cloud at any moment. Recall that WAL objects written before a checkpoint are deleted from the cloud as soon as the checkpoint is completely uploaded. Consequently, the amount of storage is directly proportional to the number of updates per minute (W – assuming each update uses a record), and to the $Ckpt_{Time}$, which includes the checkpoint period, its duration, and the amount of time that it takes to be uploaded to the cloud.

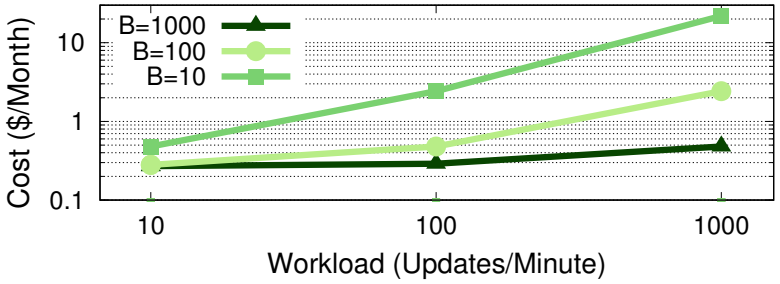


Figure 9.4: Effect of different configurations and workloads in Ginja's monetary cost for a 10GB-database and Amazon S3.

The total number of updates performed between checkpoints is divided by the number of records per WAL page (*RecPerPage*), as we coalesce multiple writes to the same page, reaching the number of WAL segments uploaded to the cloud. The "+1" considers the worst case scenario – the situation in which the first WAL write after a checkpoint is performed in the last record of a WAL segment.
Finally, $PageSize$ is the size in GB of each WAL page, CR is the compression rate and $C_{Storage}$ is the storage cost.

9.7.1.4 PUT operations of WAL objects.

Finally, the cost associated with the number of PUT operations of WAL cloud objects is represented by C_{WAL_PUT}. This cost depends essentially on the database workload and the value of the parameter B, and it is given by the following equation:

$$C_{WAL\_PUT} = \frac{W \times 60 \times 24 \times 30}{B} \times C_{PUT}$$

Every time B database updates are executed in the DBMS, a WAL object is uploaded to the cloud. Thus, the C_{WAL_PUT} is calculated using the number of database updates executed per month and multiplying this value by the price charged for each PUT operation.

9.7.2 The Cost of Running Ginja

Figure 9.4 presents the operational monetary costs of Ginja with different values of B and under different workloads. The values presented consider the usage of Amazon S3, and a database of 10GB with pages of 8kB containing 75 WAL records. We also consider that a checkpoint happens every 60 minutes, has a duration of 20 minutes, and a compression rate of 1.43 (i.e., every 1MB becomes 700kB).
The results show that the parameter B has a severe impact on the total monetary cost of Ginja. This can be explained by the fact that B reduces the number of executed cloud synchronizations (i.e., PUT operations). Additionally, we can also observe that this relation is even more evident when considering more demanding update-heavy workloads.
It is worth to mention that the size of the database we consider (10GB) implies in a fixed $C_{DB\_Storage}$ of $0.26. If one wants to consider, for instance, a 10× bigger database, the cost will be $2.62.
These results show that there are plenty of possible configurations that cost less than $1 per month. For reference, the cheapest VM in Amazon EC2 (Linux t1.micro, with 600MB of memory) costs $14.64/month in October 2016.

9.7.2.1 Real application.

We now present an evaluation of the costs of Ginja considering the database used in a real clinical analysis system deployed in more than 100 institutions in Europe. Table 9.2 presents the monetary

Configuration	Ginja with S3	EC2 VMs
Laboratory (10GB, 6 up/min)	**$0.48** (1 sync./min) **$1.56** (6 sync./min)	VM t2.small + VPN + EBS 100IOS = **$61.6**
Hospital (1TB, 138 up/min)	**$26.5** (1 sync./min) **$27.5** (6 sync./min)	VM t2.medium + VPN + EBS 500IOS = **$190.6**

Table 9.2: Costs of performing cloud-based disaster recovery with AWS using Ginja or database replication with VMs.

costs of performing disaster recovery in the cloud (specifically, Amazon Web Services) using Ginja with one (RPO $\approx$ 1 minute) and six (RPO $\approx$ 10 seconds) cloud synchronizations per minute. For comparison purposes, the table also shows the cost of a DR solution based on a single backup database VM on Amazon EC2, as a pilot light for recovering the system [271].[4] We consider two database configurations: one hospital with a 1TB-database and a workload of 630 transactions per minute, and a clinical laboratory with a 10GB-database that processes 30 transactions per minute. Among these transactions, only 20% are updates. These results are averages obtained through a month.
In the laboratory scenario, Ginja has an operational cost between 39× to 128× smaller when compared with the cost of using a backup replica in a VM. The dominant factor in this scenario is the cost of uploading WAL objects to the cloud, i.e., C_{WAL_PUT}. In the hospital scenario, Ginja has a cost 7× smaller than the cost of running a backup database on a VM instance in the cloud. The benefits of Ginja in this case are not so expressive as the cost is dominated by the storage of the DB objects (1.25TB, on average).
These results show that using Ginja is substantially cheaper than maintaining VM instances in the cloud, especially for small to medium databases, which are expected to be the norm in SMEs. Perhaps even more importantly, besides these economical advantages, our system is arguably much simpler to manage than the alternative solution, which requires configuring a firewall, setting up a public IP, etc.

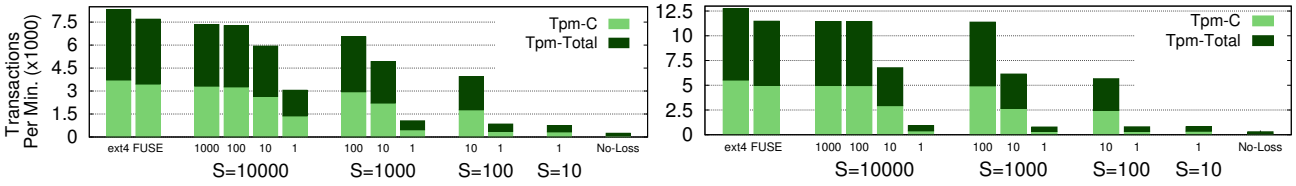
9.7.3 The Cost of Recovery

The cost of recovering a database backed-up using Ginja is basically defined by the cost of downloading all DB and WAL objects. Currently, the costs of downloading a GB of data is 3× higher than the cost of storing it for a month in Amazon S3 [3]. Therefore, the cost of recovering a database can be approximated by $3 \times (C_{DB\_Storage} + C_{WAL\_Storage})$ plus the costs of the GET operations used to download these files (not significant). For instance, the costs of recovering from a disaster on the real clinical databases mentioned before would be $112.5 and $1.125 for the Hospital and the Laboratory, respectively. Importantly, if the database is recovered to a EC2 VM in the same location as the data, this cost goes to zero, as downloads from S3 to EC2 in the same region are free of charge [3].

9.8 Experimental Evaluation

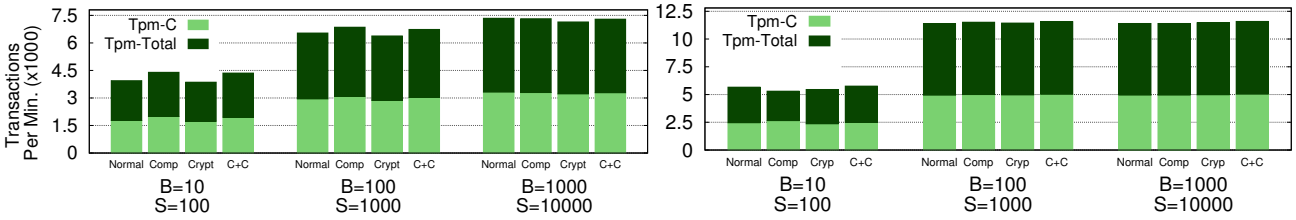
In this section we present an experimental evaluation of Ginja using the PostgreSQL 9.3 database management system [19] and MySQL 5.7 with InnoDB [17]. The experiments were executed in two Dell Power Edge R410 machines (one for the DBMS and Ginja and another for the benchmark software) equipped with two Intel Xeon E5520 CPUs (quad-core, HT, 2.27Ghz), 32GB of RAM and a 146GB Hard Disk Drive with 15k RPMs. The operating system used was Ubuntu Server (14.04 LTS, 64-bits), with kernel 3.5.0-23-generic and Java 1.8.0 (64-bits). The databses' default configuration was used in all experiment. The cloud storage service used was Amazon S3 (US Standard).
We report average results from five executions running TPC-C [1] during 5 minutes. We chose this benchmark for measuring the overhead of Ginja due to its update-heavy workload ($\approx$ 90% of updates), as our system has no effect on read transactions. For PostgreSQL we used the BenchmarkSQL 4.1.1

[4] Values obtained using `https://calculator.s3.amazonaws.com`.



(a) PostgreSQL (b) MySQL

Figure 9.5: Influence of different configurations in the performance of Ginja with PostgreSQL and MySQL. The values of B are expressed immediately below the columns. Exceptions are the first two columns (native file system and FUSE), and the last column ($S = B = 1$).



(a) PostgreSQL (b) MySQL

Figure 9.6: Effect of compression and cryptography in the performance of Ginja. The columns are grouped by configuration (B and S), and the values immediately below de columns specify whether compression, cryptography or both (C+C) are active.

tool [4] with one warehouse and 5 terminals, while for MySQL we used a Java implementation of TPC-C [12] configured with two warehouses and 60 terminals.[5] The reported metrics are the total number of transactions per minute (Tpm-Total), and the number of newOrder transactions per minute while the DBMS is also processing other types of transactions (Tpm-C). In all experiments Ginja was configured with five Uploader threads, which corresponds to the best setup in our environment.

9.8.1 Overhead of Ginja

9.8.1.1 Performance overhead.

Figure 9.5 shows the effect that different configurations of B (Batch) and S (Safety) have in the throughput of the PostgreSQL and MySQL running TPC-C on top of Ginja. We also ran the benchmark on top of a native file system (ext4) and of a FUSE file system[6] in order to have a baseline for comparison.

The first observation to make is that the FUSE-J file system presents a throughput decrease of 7% and 12% for PostgreSQL and MySQL, respectively. Since Ginja is also a FUSE-J file system, this will be our baseline.

The most important observation is that, for sufficiently high values of B and S, Ginja introduces a small performance loss (3.7% and 1.1% for PostgreSQL and MySQL, respectively). Furthermore, small values of B make the amount of pending updates reach S earlier, constantly blocking the DBMS and decreasing its performance.

The figure also shows results for Ginja with $S = B = 1$ (No Loss), which corresponds to synchronous replication. As expected, this configurations presents the lowest performance among all the ones we tested: 248 and 348 Tpm-Total, for PostgreSQL and MySQL, respectively.

[5] We chose these configurations as they allow the DBMS to reach the highest performance without Ginja.

[6] It only catches the DBMS disk accesses and writes them locally.

Configuration	Num. PUTs (5 min)		Object Size (kB)		PUT latency (millisec.)	
	PG	MS	PG	MS	PG	MS
10/100 plain	1789	3864	386	26	692	391
10/100 C+C	1990	3994	237	11	562	376
100/1000 plain	364	1046	3018	180	2880	698
100/1000 C+C	383	1063	1908	78	2007	610
1000/10000 plain	119	139	10081	1309	7707	1552
1000/10000 C+C	119	137	6339	606	4422	1354

Table 9.3: Ginja's use of storage cloud. All results are averages collected during five executions of five minutes of TPC-C for different configurations with both PostgreSQL (PG) and MySQL (MS).

Configuration	PostgreSQL		MySQL	
	CPU	Memory	CPU	Memory
Native FS	6.4%	4.3%	13.7%	1.2%
FUSE FS	6.9%	4.9%	14.9%	1.7%
100/1000	7.8%	6.9%	15.3%	8.1%
100/1000 Comp	11.6%	9.7%	15.8%	12.1%
100/1000 Crypt	9.1%	7.2%	16.4%	9.7%
100/1000 C+C	13.4%	9.9%	16.0%	11.1%

Table 9.4: Database server (eight cores with hyper-threading and 32GB of RAM) resource usage with and without Ginja.

9.8.1.2 Compression and encryption.

Figure 9.6 shows how compression and encryption influence the performance of Ginja. For PostgreSQL (Figure 9.6a), the use of these features made the results vary slightly, as the latency of uploading compressed data is smaller (see next section). On the other hand, encryption introduces a minimal overhead. For MySQL (Figure 9.6b), there are basically no changes in performance. This happens because the page size of MySQL WAL segments are quite small (512 bytes vs. 8kB in PostgreSQL), leading to diminished effects of compression and encryption in the data upload latency.

9.8.2 Resource Usage

9.8.2.1 Cloud usage and its implications.

Table 9.3 shows the number of PUTs, the size of the objects written and the observed upload latency during the benchmark execution of a 5-minute TPC benchmark. We focus our discussion on PostgreSQL results, but the insights are similar for MySQL.
The results show that increasing the batch from 10 to 100 decreases the number of PUTs by 80%, while an additional tenfold increase further decreases this number by almost 70%. In the same way, increasing the batch increases the object size and, consequently, the latency to write the object to the cloud. However, this increase is not linearly proportional with the increase of the object size due to coalescing of writes during the page aggregation.
The table shows also that using compression (and encryption) reduces the object size by 37%, reducing the PUT latency, and bringing the benefits discussed before.

9.8.2.2 Database server resource usage.

Table 9.4 presents the resource usage of a database server running a TPC-C workload under different configurations with and without Ginja.
For PostgreSQL, the table shows that using a Native or FUSE file system already require around 8% of the machine CPU and less than 1.6GB of memory ($< 5\%$). When using Ginja, the server CPU and memory usage increase by 1% and 2%, respectively, when compared with a FUSE FS. Additionally,

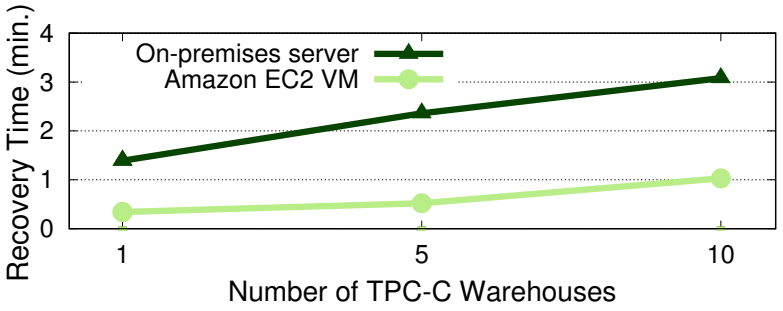


Figure 9.7: Recovery times of GINJA for different database sizes using a local server and a VM in the same location as the data.

compression and encryption introduce some CPU load: +4.5% and +1.5%, respectively. In terms of memory, these features increase the memory usage by 3% (compression) and 0.3% (encryption). When compression and encryption are used, the overheads of these features are summed up.
For MySQL, the CPU usage is basically the same, independently of the enabled features (under the standard deviation of $\approx$ 10%). The memory usage follows the same trends as in PostgreSQL: compression demands more memory than encryption.
In the end, using GINJA with compression and encryption requires at most +7% of CPU (for PostgreSQL, which is less than a core in our server) and +10% of memory (for MySQL, less than 3.2GB) of our 8-core 32GB server. We consider these costs would not be a deterrent for using GINJA.

9.8.3 Recovery Time

Our last experiment measure the recovery time of GINJA after experiencing a failure when executing TPC-C for five minutes. The experiment was done with PostgreSQL, but the results for MySQL would be similar as the key factor here is the database download time from the storage service.
We ran the experiment for three different database sizes, by varying the number of warehouses in TPC-C [1] (with a maximum DB size of 1.5GB) and executed the recovery process in a machine in our site, and in an Amazon EC2 VM (located in the same region where GINJA stored the data).
As expected, the recovery time grows with the database size as more data has to be downloaded. Furthermore, the recovery time can be remarkably reduced by executing GINJA in a computing instance located in the same cloud infrastructure where the backup data is being stored.

9.9 Related Work

9.9.1 Database disaster recovery.

There is a large body of work related with database replication for fault tolerance [195] (which is mostly orthogonal to this work), but practical off-the-shelf systems normally implement only the simplest solutions [98]. Here we discuss some relevant solutions for PostgreSQL and MySQL disaster recovery.
PostgreSQL provides two mechanisms for helping disaster tolerance [20]. The first one, Continuous Archiving, consists of performing a file-system-level backup of the database directory and setting a process (the archiver) that periodically backs up completed WAL segments. This mechanism could be used to tolerate disasters by configuring the archiver process to copy the log files to a geographically remote facility such as a cloud storage service. However, the archiver process only operates over completed WAL segments, and thus it does not provide any fine-grained control over the RPO. The second mechanism is named Streaming Replication, and allows a primary server to replicate, in a synchronous or asynchronous way, the changes made to its database to a backup server.

MySQL also offers replication solutions very similar to PostgreSQL' streaming replication, supporting asynchronous, delayed or synchronous replication [18]. In both databases, this primary-backup replication could be used as a disaster tolerance solution by placing the backup replica in a cloud VM. As discussed before, this implies substantially higher costs than what we achieve with Ginja.
PostgreSQL and MySQL can also be protected by a third-party solution named Zmanda [25], which is a bit more closer to Ginja. This tool extends the PostgreSQL and MySQL backup solutions, and improves them by allowing customers to specify a well-defined backup schedule and do point-in-time recovery in a simple way. Zmanda also allows the execution of (full or incremental) online backups to Amazon S3 or Google Storage. Since Zmanda only backs up the state of the databases at the schedule time, it can not provide a fine-grained control over the RPO as Ginja, that works at the transaction commit level. Furthermore, being a commercial service, the costs of using Zmanda are much higher than running Ginja.

9.9.2 Filesystem mirroring.

A common way of having disaster tolerance is by replicating data at the storage level. By continuously backing up the relevant files to remote storage facilities, a system is no longer susceptible to lose all its data if a disaster occurs in its primary infrastructure.
Two examples of such systems are SnapMirror [256] and Seneca [186]. Both use asynchronous replication in order to avoid any significant loss of performance. The main difference between these two solutions is that the first replicates consistent file system snapshots, while the second sends batches of updates to the remote site.
The most important advantage of such solutions is that they allow any application to protect its data, without requiring changes to its source code. However, they do not consider the semantics of the applications, which can result in inconsistent states after recovery. Additionally, these solutions require computing instances running on the backup site, which implies much higher costs in the cloud than running Ginja.

9.9.3 Virtual machine replication.

The virtualization of IT resources is one of the key features of modern disaster tolerance/recovery strategies [271, 316]. Here we discuss some works for transparent VM replication that could be used for disaster recovery in database systems.
RemusDB [237] is an extension for the Remus VM replication system [122] that provides high availability for DBMSs in a transparent manner. This is achieved by running the DBMS in a virtual machine, making the virtualization layer perform the high availability tasks related with data replication, failure detection and recovery. A key limitation of RemusDB is that it was not designed for wide-area replication, and the higher latencies can render the system impractical.
SecondSite [267] is another extension for Remus, specifically designed for disaster recovery. The system continuously replicates the entire state of several virtual machines to backup images in a different geographic location, which can assume the responsibility after a disaster in the primary site in a completely transparent manner. SecondSite deals with the limitations of wide-area replication by making a better use of bandwidth through checkpoint compression, and using quorums of servers for detecting failures.
PipeCloud [317] is a cloud-based disaster recovery system for multi-tier client-server applications running on a set of VMs. This system runs in the virtual machine monitor of each cloud physical server and replicates all disk writes to geographically distant backup servers.
All these virtualization-based approaches have the advantage of performing fast failover since they include a backup VM running in a secondary site ready to take over when a disaster is detected in the primary infrastructure. Such additional computing resources implies higher operational costs.

9.9.4 Cloud-backed storage services.

Although the following solutions were not explicitly conceived for disaster recovery, the mechanisms they employ are often similar to the ones we use in Ginja.
Brantner et al. [75] presented a DBMS core design that uses Amazon S3 as its storage subsystem. This core allows retrieving pages from S3, buffering them locally (in memory or disk), updating them, and then writing them back. All this remote operations are coordinated by a page manager, on top of which there is a record manager that provides a record-oriented interface to the applications. The work proposes several protocols for accessing the cloud services with different guarantees, but its design does not prioritizes either cost or performance. Furthermore, integrating this solution with existing DBMS requires substantial reengineering effort, on the contrary of Ginja.
Cumulus [306] is a utility that performs efficient file system backups to cloud storage services. Thus, it can be used to take snapshots of the data directory where DBMSs writes preventing the failure of the local infrastructure.
Cloud-backed file systems such as BlueSky [307] and SCFS [66] translate local file system operations to a cloud storage service with minimum or no use of cloud VMs. SCFS in particular provides strong consistency and durability guarantees and thus could be used for implementing disaster recovery on a database running on top of it. However, the system implements only synchronous or asynchronous replication of whole files, which means that the database files replication will be very inefficient.

9.10 Conclusion

We presented Ginja, a transactional DBMS disaster recovery system that uses the public cloud storage services for offering efficient and low-cost DR. Our current prototype supports PostgreSQL and MySQL, and the experimental results show that using our system degrades the database performance by less than 5% when running TPC-C, with less than 10% additional CPU and memory load on our server. Furthermore, our system is between 7-108× cheaper than having a VM-based cloud disaster recovery service for a database used in a real application.

Part III

Advanced privacy-preserving components

Chapter 10 Privacy-Preserving Outsourcing by Distributed Verifiable Computation

Verifiable computation allows a client to outsource computations to a worker with a cryptographic proof of correctness of the result that can be verified faster than performing the computation. Recently, the Pinocchio system achieved faster verification than computation in practice for the first time. Unfortunately, Pinocchio and other efficient verifiable computation systems require the client to disclose the inputs to the worker, which is undesirable for sensitive inputs for preserving the privacy of the owners of the data.
In this chapter, we propose the Trinocchio system for solving this problem. This work builds on the preliminary architecture of the SUPERCLOUD data management and storage presented in deliverable D3.1 "Architecture for Data Management"[308]. More specifically, it advances the state-of-the-art solutions regarding privacy enabling mechanisms like secure multi-party computation (MPC), secret sharing and verifiable computation.
Trinocchio is a system that distributes Pinocchio to three (or more) workers, that each individually does not learn which inputs they are computing on. We fully exploit the almost linear structure of Pinocchio proofs, letting each worker essentially perform the work for a single Pinocchio proof; verification by the client remains the same. We created a SUPERCLOUD prototype for this approach and integrated it within the SUPERCLOUD architecture.

10.1 Introduction

Recent cryptographic advances are starting to make verifiable computation more and more practical. The goal of verifiable computation is to allow a client to outsource a computation to a worker and cryptographically verify the result with less effort than performing the computation itself. Based on recent ground-breaking ideas [168, 159], Pinocchio [255] was the first implemented system to achieve this for some realistic computations. Recent works have improved the state-of-the-art in verifiable computation, e.g., by considering better ways to specify computations [63], or adding access control [33].
However, one feature not yet available in practical verifiable computation is privacy, meaning that the worker should not learn the inputs that it is computing on. This feature would enable a client to save time by outsourcing computations, even if the inputs of those computations are so sensitive that it does not want to disclose them to the worker. In addition, it would allow verifiable computation to be used in settings where multiple clients do not trust the worker or each other, but still want to perform a joint computation over their respective inputs and be sure of the correctness of the result.
While privacy was already defined in the first paper to formalize verifiable computation [158], it has not been shown so far how it is efficiently achieved. Indeed, existing constructions rely on inefficient cryptographic primitives. By outsourcing a computation to multiple workers using multiparty computation, it *is* possible to guarantee privacy (if not all workers are corrupted) and correctness, but existing constructions from the literature lose the most appealing feature of verifiable computation: namely, that the computations can be verified very quickly, even in time independent from the computation size. This leads to the question: can we perform verifiable computation with the *correctness* and *performance* guarantees of [255], but while also getting *privacy* against corrupted workers?

10.2 Related Work

Privacy-preserving outsourcing to single workers has been considered in the literature, but constructions in this setting rely on inefficient cryptographic primitives like fully homomorphic encryption [158, 108, 147], functional encryption [164], and multi-input attribute-based encryption [167]. (This is not surprising: indeed, even without guaranteeing correctness, letting a single worker perform a computation on inputs it does not know would intuitively seem to require some form of fully homomorphic encryption.) Some of these works also consider a multi-client setting [108, 167].
A large body of works considers multiparty computation for privacy-preserving outsourcing (see, e.g., [190, 259, 95, 184]). These works do not consider verifiability and achieve correctness at best in the case that *all-but-one* workers are corrupt (due to inherent limitations of the underlying protocols). We stress that this is rather unsatisfactory for the outsourcing scenario, where one naturally wishes to cover the case that *all* workers are corrupt—dispensing of the need to trust any particular worker.
Concerning outsourcing to multiple workers, [38] presents a verifiable computation protocol combining privacy and correctness; but unfortunately, they guarantee neither privacy nor correctness if all workers are corrupted and may collude; and it places a much higher burden on the workers than, e.g., [255]. Alternatively, recent works [55, 127, 280], guarantee correctness independent of worker corruption, but privacy only under some conditions. Our work offers a substantial performance improvement over these works by fully exploiting a set-up that needs to be trusted both for guaranteeing privacy and for guaranteeing correctness.
We should mention that the notion of verifiability exists in various forms and the field has a richer background than presented here. However, we focus entirely on the notion of verifiable computation first formalized by [158], because it is tailored to the outsourcing scenario.

10.3 Distributing the Prover Computation

In this section, we present the single-client version of our Trinocchio protocol for privacy-preserving outsourcing. In Trinocchio, a client distributes computation of a function $x_2 = f(x_1)$ to n workers (we consider here single-valued input and output, but the generalisation is straightforward). Trinocchio guarantees correct function evaluation (regardless of corruptions) and secure function evaluation (if at most θ workers are passively corrupted, where $n = 2\theta + 1$). Trinocchio in effect distributes the proof computation of Pinocchio; the number of workers to obtain privacy against one semi-honest worker is three, hence its name.

10.3.1 Multiparty Computation using Shamir Secret Sharing

To distribute the Pinocchio computation, Trinocchio employs multiparty computation techniques based on Shamir secret sharing [62]. Recall that in (θ, n) Shamir secret sharing, a party shares a secret s among n parties so that $\theta + 1$ parties are needed to reconstruct s. It does this by taking a random degree-$\leq \theta$ polynomial $p(x) = \alpha_\theta x^\theta + \ldots + \alpha x + s$ with s as constant term and giving $p(i)$ to party i. Since $p(x)$ is of degree at most θ, $p(0)$ is completely independent from any θ shares but can be easily computed from any $\theta + 1$ shares by Lagrange interpolation. We denote such a sharing as $[\![s]\!]$. Note that Shamir-sharing can also be done "in the exponent", e.g., $[\![\langle a \rangle_1]\!]$ denotes a Shamir sharing of $\langle a \rangle_1 \in \mathbb{G}_1$ from which $\langle a \rangle_1$ can be computed using Lagrange interpolation in $\mathbb{G}_1$.
Shamir secret sharing is linear, i.e., $[\![a + b]\!] = [\![a]\!] + [\![b]\!]$ and $[\![\alpha a]\!] = \alpha [\![a]\!]$ can be computed locally. When computing the product of $[\![a]\!]$ and $[\![b]\!]$, each party i can locally multiply its points $p_a(i)$ and $p_b(i)$ on the random polynomials p_a and p_b. Because the product polynomial has degree at most 2θ, this is a $(2\theta, n)$ sharing, which we write as $[a \cdot b]$ (note that reconstructing the secret requires $n = 2\theta + 1$ parties). Moreover, the distribution of the shares of $[a \cdot b]$ is not independent from the values of a and b, so when revealed, these shares reveal information about a and b. Hence, in multiparty computation, $[a \cdot b]$ is typically converted back into a random (θ, n) sharing $[\![a \cdot b]\!]$ using an interactive protocol due

to [160]. Interactive protocols for many other tasks such as comparing two shared value also exist (see, e.g., [126]).

10.3.1.1 The Trinocchio protocol

We now present the Trinocchio protocol. Trinocchio assumes that Pinocchio's KeyGen [279] has been correctly performed: formally, Trinocchio works in the KeyGen-hybrid model. Furthermore, Trinocchio assumes pairwise private, synchronous communication channels. To obtain $x_2 = f(x_1)$, a client proceeds in four steps:

- The client obtains the verification key, and the workers obtain the evaluation key, using hybrid calls to KeyGen.
- The client secret shares $[\![x_1]\!]$ of its input to the workers.
- The workers use multiparty computation to compute secret-shares $[\![x_2]\!]$ of the output and $[\![\langle V_{\text{mid}}\rangle_1]\!]$, $[\![\langle \alpha_v V_{\text{mid}}\rangle_1]\!]$, $[\![\langle W_{\text{mid}}\rangle_2]\!]$, $[\![\langle \alpha_w W_{\text{mid}}\rangle_1]\!]$, $[\![\langle Y_{\text{mid}}\rangle_1]\!]$, $[\![\langle \alpha_y Y_{\text{mid}}\rangle_1]\!]$, $[\![\langle Z\rangle_1]\!]$, $[\langle H\rangle_1]$ of the Pinocchio proof, as we explain next; and sends these shares to the client.
- The client recombines the shares into $\langle V_{\text{mid}}\rangle_1$, $\langle \alpha_v V_{\text{mid}}\rangle_1$, $\langle W_{\text{mid}}\rangle_2$, $\langle \alpha_w W_{\text{mid}}\rangle_1$, $\langle Y_{\text{mid}}\rangle_1$, $\langle \alpha_y Y_{\text{mid}}\rangle_1$, $\langle Z\rangle_1$, $\langle H\rangle_1$ by Lagrange interpolation, and accepts x_2 as computation result if Pinocchio's Verify returns success.

Algorithm 8: Trinocchio's compute protocol

1: ▷ $\mathcal{S} = \{\alpha_1, \ldots, \alpha_d\}$ denotes the list of roots of the target polynomial of the QAP
2: ▷ $\mathcal{T} = \{\beta_1, \ldots, \beta_d\}$ denotes a list of distinct points different from $\mathcal{S}$
3: **function** Compute($\mathsf{EK}_f = \{\langle r_v v_i\rangle_1\}_i, \ldots, \{\langle s^j\rangle_1\}_j; [\![x_1]\!]$)
4: $([\![x_2]\!], \ldots, [\![x_k]\!]) \leftarrow f([\![x_1]\!])$
5: $[\![\boldsymbol{v}]\!] \leftarrow \{\sum_i v_i(\alpha_j) \cdot [\![x_i]\!]\}_j$; $[\![\boldsymbol{V}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\boldsymbol{v}]\!])$; $[\![\boldsymbol{v'}]\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\boldsymbol{V}]\!])$
6: $[\![\boldsymbol{w}]\!] \leftarrow \{\sum_i w_i(\alpha_j) \cdot [\![x_i]\!]\}_j$; $[\![\boldsymbol{W}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\boldsymbol{w}]\!])$; $[\![\boldsymbol{w'}]\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\boldsymbol{W}]\!])$
7: $[\![\boldsymbol{y}]\!] \leftarrow \{\sum_i y_i(\alpha_j) \cdot [\![x_i]\!]\}_j$; $[\![\boldsymbol{Y}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\boldsymbol{y}]\!])$; $[\![\boldsymbol{y'}]\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\boldsymbol{Y}]\!])$
8: $[\boldsymbol{h'}] \leftarrow \{([\![\boldsymbol{v'}_j]\!] \cdot [\![\boldsymbol{w'}_j]\!] - [\![\boldsymbol{y'}_j]\!])/t(\beta_j)\}_j$; $[\boldsymbol{H}] \leftarrow \mathsf{FFT}_{\mathcal{T}}^{-1}([\boldsymbol{h'}])$
9: $[\![\langle V_{\text{mid}}\rangle_1]\!] \leftarrow \sum_i \langle r_v v_i\rangle_1 \cdot [\![x_i]\!]$
10: $[\![\langle \alpha_v V_{\text{mid}}\rangle_1]\!] \leftarrow \sum_i \langle r_v \alpha_v v_i\rangle_1 \cdot [\![x_i]\!]$
11: $[\![\langle W_{\text{mid}}\rangle_2]\!] \leftarrow \sum_i \langle r_w w_i\rangle_2 \cdot [\![x_i]\!]$
12: $[\![\langle \alpha_w W_{\text{mid}}\rangle_1]\!] \sum_i \langle r_w \alpha_w w_i\rangle_1 \cdot [\![x_i]\!]$
13: $[\![\langle Y_{\text{mid}}\rangle_1]\!] \leftarrow \sum_i \langle r_y y_i\rangle_1 \cdot [\![x_i]\!]$
14: $[\![\langle \alpha_y Y_{\text{mid}}\rangle_1]\!] \leftarrow \sum_i \langle r_y \alpha_y y_i\rangle_1 \cdot [\![x_i]\!]$
15: $[\![\langle Z\rangle_1]\!] \leftarrow \sum_i \langle r_v \beta v_i + r_w \beta w_i + r_y \beta y_i\rangle_1 \cdot [\![x_i]\!]$
16: $[\langle H\rangle_1] = \sum_j \langle s^j\rangle_1 \cdot [\boldsymbol{H}_j]$
17: **return** $([\![x_2]\!]; [\![\langle V_{\text{mid}}\rangle_1]\!], [\![\langle \alpha_v V_{\text{mid}}\rangle_1]\!], [\![\langle W_{\text{mid}}\rangle_2]\!], [\![\langle \alpha_w W_{\text{mid}}\rangle_1]\!],$
18: $[\![\langle Y_{\text{mid}}\rangle_1]\!], [\![\langle \alpha_y Y_{\text{mid}}\rangle_1]\!], [\![\langle Z\rangle_1]\!], [\langle H\rangle_1])$

Algorithm 8 shows in detail how the secret-shares of the function output and Pinocchio proof are computed. The first step is to compute function output $x_2 = f(x_1)$ and values $(x_3, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of the QAP (line 4). This is done using normal multiparty computation protocols based on secret sharing. If function f is represented by an arithmetic circuit, then it is evaluated using local addition and scalar multiplication, and the multiplication protocol from [160]. If f is represented by a circuit using more complicated gates, then specific protocols may be used. Any protocol can be used as long as it guarantees privacy, i.e., the view of any θ workers is statistically independent from the values represented by the shares.

The next task is to compute, in secret-shared form, the coefficients of the polynomial $h = ((\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i))/t \in \mathbb{F}[x]$ that we need for proof element $\langle H \rangle_1$. In theory, this computation could be performed by first computing shares of the coefficients of $(\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$, and then dividing by t, which can be done locally using traditional polynomial long division. However, this scales quadratically in the degree of the QAP and hence leads to unacceptable performance. Hence, we take the approach based on fast Fourier transforms (FFTs) from [63], and adapt it to the distributed setting. Given a list $\mathcal{S} = \{\omega_1, \ldots, \omega_d\}$ of distinct points in $\mathbb{F}$, we denote by $\mathbf{P} = \mathsf{FFT}_{\mathcal{S}}(\mathbf{p})$ the transformation from coefficients $\mathbf{p}$ of a polynomial p of degree at most $d-1$ to evaluations $p(\omega_1), \ldots, p(\omega_d)$ in the points in $\mathcal{S}$. We denote by $\mathbf{p} = \mathsf{FFT}_{\mathcal{S}}^{-1}(\mathbf{P})$ the inverse transformation, i.e., from evaluations to coefficients. Deferring specifics to later, we mention now that the FFT is a linear transformation that, for some $\mathcal{S}$, can be performed locally on secret shares in $\mathcal{O}(d \cdot \log d)$.

With FFTs available, we can compute the coefficients of h by evaluating h in d distinct points and applying FFT^{-1}. Note that we can efficiently compute evaluations $\mathbf{v}$ of $v = (\sum_i x_i v_i)$, $\mathbf{w}$ of $w = (\sum_i x_i w_i)$, and $\mathbf{y}$ of $y = (\sum_i x_i y_i)$ in the zeros $\{\omega_1, \ldots, \omega_d\}$ of the target polynomial. Namely, the values $v_k(\omega_i)$, $w_k(\omega_i)$, $y_k(\omega_i)$ are simply the coefficients of the quadratic equations represented by the QAP, most of which are zero, so these sums have much fewer than k elements (if this were not the case, then evaluating v, w, and y would take an unacceptable $O(d \cdot k)$). Unfortunately, we cannot use these evaluations directly to obtain evaluations of h, because this requires division by the target polynomial, which is zero in exactly these points ω_i. Hence, after determining $\mathbf{v}$, $\mathbf{w}$, and $\mathbf{y}$, we first use the inverse FFT to determine the coefficients $\mathbf{V}$, $\mathbf{W}$, and $\mathbf{Y}$ of v, w, and y, and then again the FFT to compute the evaluations $\mathbf{v}'$, $\mathbf{w}'$, and $\mathbf{y}'$ of v, w, and y in another set of points $\mathcal{T} = \{\Omega_1, \ldots, \Omega_k\}$ (lines 5–7). Now, we can compute evaluations $\mathbf{h}'$ of h in $\mathcal{T}$ using $h(\Omega_i) = (v(\Omega_i) \cdot w(\Omega_i) - y(\Omega_i))/t(\Omega_i)$. This requires a multiplication of (θ, n)-secret shares of $v(\Omega_i)$ and $w(\Omega_i)$, hence the result is a $(2\theta, n)$-sharing. Finally, the inverse FFT gives us a $(2\theta, n)$-sharing of the coefficients $\mathbf{H}$ of h (line 8). Given secret shares of the values of x_i and coefficients of h, it is straightforward to compute secret shares of the Pinocchio proof. Indeed, $\langle V_{\text{mid}} \rangle_1, \ldots, \langle H \rangle_1$ are all computed as linear combinations of elements in the evaluation key, so shares of these proof elements can be computed locally (lines 9–16) , and finally returned by the respective workers (lines 17–18).

Note that compared to Pinocchio, our client needs to carry out slightly more work. Namely, our client needs to produce secret shares of the inputs and recombine secret shares of the outputs; and it needs to recombine the Pinocchio proof. However, according to the micro-benchmarks from [255], this overhead is small. For each input and output, Verify includes three exponentiations, whereas Combine involves four additions and two multiplications; when using [255]'s techniques, this adds at most a 3% overhead. Recombining the Pinocchio proof involves 15 exponentiations at around half the cost of a single pairing.

Alternatively, it is possible to let one of the workers perform the Pinocchio recombining step by using the distributed zero-knowledge variant of Pinocchio and the techniques from Section 10.4. In this case, the only overhead for the client is the secret-sharing of the inputs and zero-knowledge randomness, and recombining the outputs.

10.3.1.1.1 Parameters for Efficient FFTs

To obtain efficient FFTs, we use the approach of [63]. There, it is noted that the operation $\mathbf{P} = \mathsf{FFT}_{\mathcal{S}}(\mathbf{p})$ and its inverse can be efficiently implemented if $\mathcal{S} = \{\omega, \omega^2, \ldots, \omega^d = 1\}$ is a set of powers of a primitive dth root of unity, where d is a power of two. (We can always demand that QAPs have degree $d = 2^k$ for some k by adding dummy equations.) Moreover, [63] presents a pair of groups $\mathbb{G}_1, \mathbb{G}_2$ of order q such that $\mathbb{F}_q$ has a primitive 2^{30}th root of unity (and hence also primitive 2^kth roots of unity for any $k < 30$) as well as an efficiently computable pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_3$. Finally, [63] remarks that for $\mathcal{T} = \{\eta\omega, \eta\omega^2, \ldots, \eta\omega^d = \eta\}$, operations $\mathsf{FFT}_{\mathcal{T}}^{-1}$ and $\mathsf{FFT}_{\mathcal{T}}^{-1}$ can easily be reduced to $\mathsf{FFT}_{\mathcal{S}}$ and $\mathsf{FFT}_{\mathcal{S}}^{-1}$, respectively. In our implementation, we use exactly these suggested parameters.

10.3.1.2 Security of Trinocchio

Theorem 6. *Let f be a function. Let $n = 2\theta + 1$ be the number of workers used. Let d be the degree of the QAP computing f used in the Trinocchio protocol. Assuming the d-PKE, $(4d + 4)$-PDH, and $(8d + 8)$-SDH assumptions:*

- *Trinocchio correctly evaluates [279] f in the* KeyGen*-hybrid model.*
- *Whenever at most θ workers are passively corrupted, Trinocchio securely evaluates f in the* KeyGen*-hybrid model.*

The proof of this theorem is easily derived as a special case of the proof for the multi-client Trinocchio protocol later. Here, we present a short sketch.
To prove correct function evaluation, we need to show that for every real-world adversary $\mathcal{A}$ interacting with Trinocchio, there is an ideal-world simulator $\mathcal{S}_{\mathcal{A}}$ that interacts with the trusted party for correct function evaluation such that the two executions give indistinguishable results. The only interesting case is when the client is honest and some of the workers are not. In this case, the simulator receives the input of the honest party, and needs to choose whether to provide the output. To this end, the simulator simply simulates a run of the actual protocol with $\mathcal{A}$, until it has finally obtained function output x_2 and the accompanying Trinocchio proof. If the proof verifies, it tells the trusted party to provide the output to the client; otherwise, it tells the trusted party not to. Finally, the simulator outputs whatever $\mathcal{A}$ outputs. Because Trinocchio is secure, except with negligible probability a verifying proof implies that the real-world output of the client (as given by the adversary) matches the ideal-world output of the client (as computed by the trusted party); and by construction, the outputs of $\mathcal{A}$ and $\mathcal{S}_{\mathcal{A}}$ are distributed identically. This proves correct function evaluation.
For secure function evaluation, again the only interesting case is if the client is honest and some of the workers are passively corrupted. In this case, because corruption is only passive, correctness of the multiparty protocol used to compute f and correctness of the Pinocchio proof system used to compute the proof together imply that real-world executions (like ideal-world executions) result in the correct function result and a verifying proof. Hence, we only need to worry about how $\mathcal{S}_{\mathcal{A}}$ can simulate the view of $\mathcal{A}$ on the Trinocchio protocol without knowing the client's input. However, note that the workers only use a multiparty computation to compute f (which we assume can be simulated without knowing the inputs), after which they no longer receive any messages. Hence simulating the multiparty computation for f and receiving any messages that $\mathcal{A}$ sends is sufficient to simulate $\mathcal{A}$. This proves secure function evaluation.

10.3.1.2.1 Privacy against Active Attacks

We remark that, actually, Trinocchio in some cases provides privacy against corrupted workers as well. Namely, suppose that the protocol used to compute f does not leak any information to corrupted workers in the event of an active attack (even though in this case it may not guarantee correctness). For instance, this is the case for the protocol from [160]: the attacker can manipulate the shares that it sends, which makes the computation return incorrect results; but since the attacker always learns only θ many shares of any value, it does not learn any information. Because the attacker learns no additional information from producing the Pinocchio proof, the overall protocol still leaks no information to the adversary. (In addition, security of Pinocchio ensures the client notices the attacker's manipulation.) This crucially relies on the workers not learning whether the client accepts the proof. If the workers would learn whether the client obtained a validating proof, then, by manipulating proof construction, they could learn whether a modified version of the tuple $(x_1, \ldots, x_k)$ is a solution of the QAP used, so corrupted workers could learn one chosen bit of information about the inputs (cf. [240]).

10.4 Handling Mutually Distrusting In- and Outputters

We now consider the scenario where there are multiple (possibly overlapping) input and result parties. There are some significant changes between this scenario and the single-client scenario. In particular, we need to extend Pinocchio to allow verification not based on the actual input/output values (indeed, no party sees all of them) but on some kind of representation that does not reveal them. Moreover, we need to use the zero-knowledge variant of Pinocchio and we need to make sure that input parties choose their inputs independently from each other.

10.4.1 Multi-Client Proofs and Keys

Our multi-client Trinocchio proofs are a generalisation of the zero-knowledge variant of Pinocchio with modified evaluation and verification keys. Recall that in Pinocchio, the proof terms $\langle V_{\text{mid}}\rangle_1$, $\langle \alpha_v V_{\text{mid}}\rangle_1$, $\langle W_{\text{mid}}\rangle_2$, $\langle \alpha_w W_{\text{mid}}\rangle_1$, $\langle Y_{\text{mid}}\rangle_1$, $\langle \alpha_y Y_{\text{mid}}\rangle_1$, and $\langle Z\rangle_1$ encode circuit values $x_{l+m+1}, \ldots, x_k$; in the zero-knowledge variant, these terms are randomised so that they do not reveal any information about $x_{l+m+1}, \ldots, x_k$. In the multi-client case, additionally, the inputs of all input parties and the outputs of all result parties need to be encoded such that no other party learns any information about them. Therefore, we extend the proof with *blocks* of the above seven terms for each input and result party, which are constructed in the same way as the seven proof terms above. Although some result parties could share a block of output values, for simplicity we assign each result party its own block in the protocol.

Algorithm 9: ProofBlock

1: **function** ProofBlock$(BK; \boldsymbol{x}; \delta_v, \delta_w, \delta_y)$
2: $\quad \langle V\rangle_1 \leftarrow \langle r_v t\rangle_1 \delta_v + \sum_i \langle r_v v_i\rangle_1 x_i$; $\langle V'\rangle_1 \leftarrow \langle r_v \alpha_v t\rangle_1 \delta_v + \sum_i \langle r_v \alpha_v v_i\rangle_1 x_i$
3: $\quad \langle W\rangle_2 \leftarrow \langle r_w t\rangle_2 \delta_w + \sum_i \langle r_w w_i\rangle_2 x_i$; $\langle W'\rangle_1 \leftarrow \langle r_w \alpha_w t\rangle_1 \delta_w + \sum_i \langle r_w \alpha_w w_i\rangle_1 x_i$
4: $\quad \langle Y\rangle_1 \leftarrow \langle r_y t\rangle_1 \delta_y + \sum_i \langle r_y y_i\rangle_1 x_i$; $\langle Y'\rangle_1 \leftarrow \langle r_y \alpha_y t\rangle_1 \delta_y + \sum_i \langle r_y \alpha_y y_i\rangle_1 x_i$
5: $\quad \langle Z\rangle_1 \leftarrow \langle r_v \beta t\rangle_1 \delta_v + \langle r_w \beta t\rangle_1 \delta_w + \langle r_y \beta t\rangle_1 \delta_y + \sum_i \langle r_v \beta v_i + r_w \beta w_i + r_y \beta y_i\rangle_1 x_j$
6: $\quad$ **return** $(\langle V\rangle_1, \langle V'\rangle_1, \langle W\rangle_2, \langle W'\rangle_1, \langle Y\rangle_1, \langle Y'\rangle_1, \langle Z\rangle_1)$

To produce a block containing values $\mathbf{x}$, a party first samples three random field values δ_v, δ_w, and δ_y and then executes ProofBlock, cf. Algorithm 9. The BK argument to this algorithm is the *block key*; the subset of the evaluation key terms specific to a single proof block. Because each input party should only provide its own input values and should not affect the values contributed by other parties, each proof block must be restricted to a subset of the wires. This is achieved by modifying Pinocchio's key generation such that, instead of a sampling a single value β, one such value, β_j, is sampled for each proof block j and the terms $\langle r_v \beta_j v_i + r_w \beta_j w_i + r_y \beta_j y_i\rangle_1$ are only included for wires indices i belonging to block j. That is, the jth block key is

$$\begin{aligned} BK_j = \{&\langle r_v v_i\rangle_1, \langle r_v \alpha_v v_i\rangle_1, \langle r_w w_i\rangle_2, \langle r_w \alpha_w w_i\rangle_1, \langle r_y y_i\rangle_1, \langle r_y \alpha_y y_i\rangle_1, \\ &\langle r_v \beta_j v_i + r_w \beta_j w_i + r_y \beta_j y_i\rangle_1, \langle r_v \beta_j t\rangle_1, \langle r_w \beta_j t\rangle_1, \langle r_y \beta_j t\rangle_1\}, \end{aligned}$$

with i ranging over the indices of wires in the block. Note that ProofBlock only performs linear operations on its $\mathbf{x}$, δ_v, δ_w and δ_y inputs. Therefore this algorithm does not have to be modified to compute on secret shares.

Algorithm 10: CheckBlock

function CheckBlock$(BV; \langle V\rangle_1, \langle V'\rangle_1, \langle W\rangle_2, \langle W'\rangle_1, \langle Y\rangle_1, \langle Y'\rangle_1, \langle Z\rangle_1)$
  if $e(\langle V\rangle_1, \langle \alpha_v\rangle_2) = e(\langle V'\rangle_1, \langle 1\rangle_2)$
   $\wedge e(\langle \alpha_w\rangle_1, \langle W\rangle_2) = e(\langle W'\rangle_1, \langle 1\rangle_2)$
   $\wedge e(\langle Y\rangle_1, \langle \alpha_y\rangle_2) = e(\langle Y'\rangle_1, \langle 1\rangle_2)$
   $\wedge e(\langle Z\rangle_1, \langle 1\rangle_2) = e(\langle V\rangle_1 + \langle Y\rangle_1, \langle \beta\rangle_2) e(\langle \beta\rangle_1, \langle W\rangle_2)$ **then**
    return $\top$
  else
    return $\bot$

A Trinocchio proof in the multi-client setting now consists of one block $\mathbf{Q}_i = (\langle V_i\rangle_1, \ldots, \langle Z_i\rangle_1)$ for each input and result party, one block $\mathbf{Q}_{\text{mid}} = (\langle V_{\text{mid}}\rangle_1, \ldots, \langle Z_{\text{mid}}\rangle_1)$ of internal wire values, and Pinocchio's $\langle H\rangle_1$ element. Verification of such a proof consists of checking correctness of each block, and checking correctness of $\langle H\rangle_1$. The validity of a proof block can be verified using CheckBlock, cf. Algorithm 10. Compared to the Pinocchio verification key, our verification key contains "block verification keys" BV_i (i.e., elements $\langle \beta_j\rangle_1$ and $\langle \beta_j\rangle_2$) for each block instead of just $\langle \beta\rangle_1$ and $\langle \beta\rangle_2$. Apart from the relations inspected by CheckBlock, one other relation is needed to verify a Pinocchio proof: the divisibility check from [279] In the protocol, the algorithm that verifies this relation will be called CheckDiv. We denote the modified setup of the evaluation and verification keys by hybrid call MKeyGen.

10.4.2 Protocol Overview

We will proceed with a protocol overview. The multi-client variant of our Trinocchio protocol makes use of private channels, just as the single-client variant, to privately communicate in- and output values, and to let the workers carry out the computation. We need some additional communication to ensure input independence and fix the input parties' values. For this, we use a bulletin board. To achieve input independence, we first have the input parties commit to a representation of their input and then reveal these, which requires the use of a commitment scheme.
Apart from key set-up there are three phases to the multi-client Trinocchio protocol.

- In the *input phase*, the input parties provide representations of their input on the bulletin board. These representations are later used as part of the proof to verify the computation results. They also serve to ensure that each input party provides its value independent of the other input values. The input parties then secret share their input values to the workers. The workers verify that the secret shared input values are consistent with their representations on the bulletin board, to prevent malicious input parties from providing a different value.

- The *computation phase* is very similar to the single-client variant of Trinocchio. In this phase, the workers perform multi-party computation to carry out the actual computation and obtain secret shares of intermediate and result wire values. They then use these secret shared wire values to construct shares of the proof elements. These are then posted on the bulletin board, instead of being communicated directly to the result parties to ensure that all result parties receive a consistent result. In order to prevent these proof elements from revealing any information about the wire values, the zero-knowledge variant of the proof is used .

- In the *result phase* the workers privately send the shares of the result values to the result parties. The result parties recombine the proof shares from the bulletin board and check whether the proof verifies. The result parties further check whether the recombined shares of the result are consistent with the information on the bulletin board. The result parties only accept the result received from the workers if both checks are satisfied.

10.4.3 Security of the Trinocchio Protocol

Analogously to the single-client case, we obtain the following result:

Theorem 7. *Let f be a function. Let $n = 2\theta + 1$ be the number of workers used. Let d be the degree of the QAP computing f used in the multi-client Trinocchio protocol. Assuming the d-PKE, $(4d+4)$-PDH, and $(8d+8)$-SDH assumptions:*

- *Trinocchio correctly evaluates f in the* $(\mathsf{ComGen}, \mathsf{MKeyGen})$*-hybrid model.*
- *Whenever at most θ workers are passively corrupted, Trinocchio securely evaluates f in the* $(\mathsf{ComGen}, \mathsf{MKeyGen})$*-hybrid model.*

We stress that "at most θ workers are passively corrupted" includes both the case when the adversary is passively corrupted, and corrupts at most θ workers (as well as arbitrarily many input and result parties); and the case when the adversary is actively corrupted, and corrupts no workers (but arbitrarily many input and result parties)
The complete proof of this theorem is given in the full version of the system [281]. To prove secure function evaluation, we obtain privacy by simulating the multiparty computation of the proof with respect to the adversary without using honest inputs. To prove correct function evaluation, we run the protocol together with the adversary: if this gives a fake Pinocchio proof, then one of the underlying problems can be broken.
In the single-client case, we remarked that Trinocchio actually provides security against up to θ *actively* corrupted workers. Namely, although θ actively corrupted workers may manipulate the computation of the function and proof, they do not learn any information from this because they do not see the resulting proof that the client gets. In our multi-client protocol, it is less natural to assume that the workers cannot see the resulting proof; and in fact, in our protocol, corrupted workers *do* see the full proof as it is posted on the bulletin board. It should be possible to obtain some privacy guarantees against actively malicious workers (who do not collude with any result parties) by letting the result parties provide proof contributions directly to the result parties instead of posting them on the bulletin board. We leave an analysis for future work.

10.5 Performance

In this section, we show that our approach indeed adds privacy to verifiable computation with little overhead. We demonstrate this in a case study: we take the "MultiVar Poly" application from [255], and show that using Trinocchio, this computation can be outsourced in a private and correct way at essentially the same cost as letting three workers each perform the Pinocchio computation.
In our experiments, one client outsources the computation to three workers. In particular, we use multiparty computation based on $(1, 3)$ Shamir secret sharing. As discussed in Sections 10.3.1.2 and 10.4.3, this guarantees privacy against one passively corrupted worker (or, in the single-client case against θ actively corrupted workers when the multiparty computation protocol does not leak any information). We did not implement the multiple client scenario; this would add small overhead for the workers, with verification effort growing linearly in the number of input and result parties but remaining small and independent from the computation size. To simulate a realistic outsourcing scenario, we distribute computations between three Amazon EC2 "m3.medium" instances[1] around the world: one in Oregon, United States; one in Ireland; and one in Tokyo, Japan. Multiparty computation requires secure and private channels: these are implemented using SSL.

10.5.1 Case Study: Multivariate Polynomial Evaluation

In [255], Pinocchio performance numbers are presented showing that, for some applications, Pinocchio verification is faster than native execution. One of these applications, "MultiVar Poly", is the evalua-

[1] Running Intel Xeon E5-2670 v2 Ivy Bridge with 4 GB SSD and 3.75 GiB RAM

	# mult	Pinoc.	Dist f	Dist π	Trinoc.	Verif.
MultiVar Poly, Medium	203428	2102	96	2092	2187	0.04
MultiVar Poly, Large	571046	6458	275	6427	6702	0.05

Table 10.1: Performance of multivariate polynomial evaluation with Trinocchio: number of multiplications in f; time for single-worker proof; time per party for computing f and proof, and total; and verification time (all times in seconds)

tion of a constant multivariate polynomial on five inputs of degree 8 ("medium") or 10 ("large"). In this case study, we use Trinocchio to add privacy to this outsourcing scenario.
We have made an implementation[2] of Trinocchio's Compute algorithm (Algorithm 8) that is split into two parts. The first part performs the evaluation of the function f (line 4) , given as an arithmetic circuit, using the secret sharing implementation of VIFF. (We use the arithmetic circuit produced by the Pinocchio compiler, hence f is exactly the same as in [255].) Note that, because f is an arithmetic circuit, this step does not leak any information against actively corrupted workers. Hence, in the single-client outsourcing scenario of Section 10.3, we achieve privacy against one actively corrupted worker. The second part is a completely new implementation of the remainder of Trinocchio using [238]'s implementation of the discrete logarithm groups and pairings from [63].
Table 10.1 shows the performance numbers of running this application in the cloud with Trinocchio. Significantly, evaluating the function f using passively secure multiparty computation (i.e., line 4 of Compute) is more than twenty times cheaper than computing the Pinocchio proof (i.e., lines 5–16 of Comp). Moreover, we see that computing the Pinocchio proof in the distributed setting takes around the same time (per party) as in the non-distributed setting. Indeed, this is what we expect because the computation that takes place is exactly the same as in the non-distributed setting, except that it happens to take place on shares rather than the actual values itself. Hence, according to these numbers, the cost of privacy is essentially that the computation is outsourced to three different workers, that each have to perform the same work as one worker in the non-private setting. Finally, as expected, verification time completely vanishes compared to computation time.

10.6 Architectural integration and prototyping

With secure multiparty computation (MPC), SUPERCLOUD servers outsource computation on their joint data to multiple workers such that no individual server sees the data it is computing on. When combining MPC with verifiable computation (VC), clients get a proof that the result of the computation is correct with respect to signed data from the servers.
Within the SUPERCLOUD system, the workers should be deployed in the compute resources of the cloud providers (as depicted in Figure 10.1). The data that is sent to the workers are the shamir shares of the input (e.g. [[s]], [[t]]) (step 1). Next, the secure multi-party computation component from each of the workers is performing the needed computations on the received input data (step 2). Within the same VM of the cloud provider, the data is transferred as shamir shares to the Pinocchio System for computing shares (e.g. [[y]], [π]) of the cryptographic proof of the computation (step 3). The cryptographic proof is generated using the evaluation key received from the pinocchio key generation component (step 0).
Later the data owner (App component), which outsourced the computation to the workers, reconstructs the results of the requested computations using the shamir secret shares that he received from the workers ([[y]], step 4). For verifying the outsourced computation, the data owner uses the verification key received from the pinocchio key generation component (step 0).
Trinocchio was prototyped within the SUPERCLOUD project. This prototype implements secure multi-party computation (MPC) and verifiable computation based on a privacy-preserving implemen-

[2]Implementation available at `http://meilof.home.fmf.nl/`

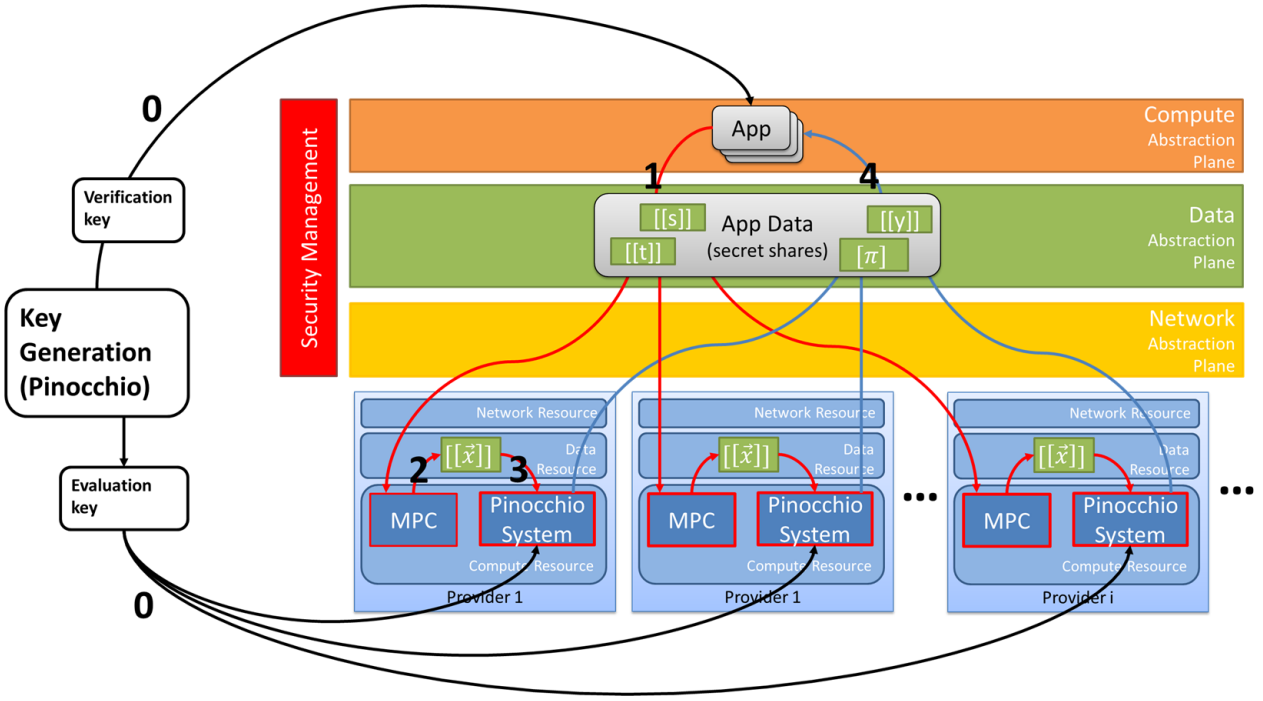


Figure 10.1: Trinocchio architectural integration

tation of Pinocchio, as depicted in Figure 10.1. The secure multi-party computation component is a deployement of the VIFF framework[3]. VIFF is a general framework for doing secure multi-party computations. In a secure multi-party computation, several parties jointly compute an agreed function without leaking any information on their inputs. This could be an election where the correct tally is computed *without* revealing any information on the individual votes. In a protocol with n players, the confidentiality of the inputs is ensured when up to n/2 of the players are corrupted. VIFF is still under development, but it is nevertheless quite usable and offers the following features:

- secret sharing based on standard Shamir and pseudo-random secret sharing (PRSS).
- arithmetic with shares from Zp or GF(2^8): addition, multiplication, exclusive-or. Some support for actively secure multiplication.
- two comparison protocols which compare secret shared Zp inputs, with secret GF(2^8) or Zp output.
- reliable broadcast, even in the presence of active adversaries.
- computations with any number of players for which an honest majority can be found.
- optional support for encrypted TLS connections between the players.

All operations are automatically scheduled to run in parallel meaning that an operation starts as soon as the operands are ready.
The computation outsourced by a client is transformed into a quadratic arithmetic program (QAP). Next this quadratic artihmetic program is used as input for both the secure multiparty computation (MPC) and pinocchio components that are run by the workers. Potentially, the C-to-QAP compiler from Pinocchio could be used to generate this QAP. However, the QAP file format of Pinocchio is currently incompatible with this prototype. We leave adding this compatibility as future work.
The first line of the QAP file format starts by giving the number of circuit values of the QAP. Then, the number of values in each "block" is given: the first block consists of public inputs and outputs;

[3] `http://security1.win.tue.nl/~meilof/files/verifiability/local.tar.gz`

each party delivering a private input or obtaining a private output should have a block; and there should be one block for the internal circuit wires of the circuit. The line ends with a dot. For instance,

```
25649 0 1640 41 41 23927 .
```

represents a QAP with 25649 circuit values: 0 public and 25649 private, divided into blocks of 1640, 41, 41, and 23927 values. (In this case, there will be 1640 inputs, 41 outputs for one party, 41 outputs for another party, and 23927 internal wire values.)
The second line specifies inputs to the QAP, starting from the first circuit value, and ending with a dot, e.g., "`1 2 3 .`". This second line is used for demo purposes by some provers; if this is not needed, then a line with a single dot suffices.
The remainder of the lines give the QAP equations $v \cdot w = y$, with v, w, and y linear functions $\alpha_1 x_1 + \cdots + \alpha_n x_n + \alpha_0$. A term $\alpha_k x_k$ is given as "α_k `k`"; the constant term as "α_0 `0`". Plus signs are ommitted, and "`*`" and "`=`" are used to separate the v, w, and y parts. E.g., "`1 1763 * 1 0 -1 1763 = .`" represents the equation $1 \cdot x_{1763} \cdot (1 + -1 \cdot x_{1763}) = 0$.
The tools available in the Trinocchio prototype are:

- `genkey` Generates evaluation and verification keys from a given QAP.
- `combine` Reads proofs and combines them into one overall proof. This combines all blocks from the parts given in the respective proof files.
- `eval` Produces a ZK-QAP proof based on the given wire values and randomness.
- `ver` Verify a QAP proof. Takes as input an evaluation key.

10.7 Conclusion

In this chapter we have presented Trinocchio, a system that adds privacy to the Pinocchio verifiable computation scheme essentially at the cost of replicating the Pinocchio proof production algorithm at three (or more) servers. Trinocchio has the same correctness and security guarantees as Pinocchio; distributing the computation between $2\theta + 1$ workers gives privacy if at most θ of them are corrupted. We have shown in a case study that the overhead is indeed small.
As far as we are aware, our work is the first to deliver efficient verifiable computation (i.e., with cryptographic guarantees of correctness and practical verification times independent of the computation size) with privacy guarantees. Although privacy is only guaranteed if not too many of the workers are corrupt, the use of verifiable computation ensures that the outcome of the protocol cannot be manipulated by the workers. This allows us to hedge against an adversary being more powerful than anticipated in a real world scenario. Existing verifiable computation constructions in the single-worker setting [158, 164, 147] use very expensive cryptography, while multiple-worker efforts to provide privacy [38] do not guarantee correctness if all workers are corrupted. In contrast, existing works from the area of multiparty computation [55, 280, 127] deliver privacy and correctness guarantees, but have much less efficient verification.
Trinocchio is a first step towards privacy-preserving verifiable computation, and there are many promising directions for future work. Recent work in verifiable computation has extended the Pinocchio approach by making it easier to specify computations [63], and by adding access control functionality [33]. In future work, it would be interesting to see how these kind of techniques can be used in the Trinocchio setting. Also, recent work has focused on applying verifiable computation on large amounts of data held by the server (and possibly signed by a third party) [105]; assessing the impact of distributing the computation (in particular when aggregating information from databases from several parties) in this scenario is also an important future direction. It would also be interesting to base Trinocchio on the (much faster) Pinocchio codebase [255] and more efficient multiparty computation implementations, and see what kind of performance improvements can be achieved. Another

interesting direction is to investigate the possibility of practical universally compose-able [92, 93] distributed verifiable computation; or to use the universal composability framework to obtain a more generic framework for combining multiparty computation with verifiable computation (even with only standalone guarantees).

Chapter 11 Privacy of Image Processing

In this chapter, we target on privacy-preserving image processing algorithms in data storage management. The concrete use cases targeted by the SUPERCLOUD project in work package 5 address the medical sector and involve the use and processing of medical images. Privacy-preserving processing techniques specifically targeted for image data are therefore of particular interest for those use cases. In Sect. 11.1 we first overview popular image processing techniques and survey privacy-preserving techniques for image processing algorithms in Sect. 11.2. In Sect. 11.3, we survey the security and performance analysis of different privacy-preserving image processing algorithms. SGX-based privacy-preserving image processing implementatio are subsequently discussed in Sect. 11.4.

11.1 Image Processing Techniques

11.1.1 Content Based Image Retrieval

Content Based Image Retrieval (CBIR) is a solution for the image retrieval problem to search for images in databases, one important application for multimedia information retrieval (MIR) [217, 285]. Indexing of images in CBIR is done by the visual content of the image [150, 273]. The main goal of CBIR system is to extract a signature or visual features from images. Features to describe the content could be like colour, texture, shape and so on. For a general CBIR scheme, the user provides a query image, and the feature vector is extracted to query the index structure. A query for CBIR scheme could be an image or a part of it. One advantage of CBIR is that the user can express the query more precisely, with smaller semantic gap between the system and the user. Moreover, CBIR systems automate the retrieval process, compared with traditional keyword-based systems. Keyword-based systems usually require time-consuming annotation of database images.

In the process of querying, the index structure is traversed, and suitable child nodes are chosen to traverse next, based on the feature vector of the query and the data in the node. Usually, based on the underlying index structure, functions, e.g., threshold functions or distance metrics are used for making the decision for the next traversing. The results are supposed to be in leaf nodes, and will be returned to the user at the end of inquiry. In some situations, we may need backtracking to acquire the complete results.

A typical CBIR system needs the construction of image descriptors, characterized by, first, an extraction algorithm which encodes image features into feature vectors; second, similarity metric to measure and compare two images. The function of similarity measure provides the degree of similarity for a pair of images, based on their feature vectors. Usually, this could be a distance metric (e.g., Euclidean). The larger the distance value, the less similar the images are. Therefore, an image descriptor is a pair of functions: an extraction function for feature vector and a distance function. Existing image descriptors include color descriptors, texture descriptors, shape descriptors and so on.

Content Based Image Retrieval is widely used in, e.g., scientific databases, general image collections for licensing, biodiversity information systems, medical image systems, fingerprint identification, digital libraries, historical research, and so on. For example, a large amount of images in medical systems are produced and stored in an outsourced cloud. Traditional CBIR schemes focus on how to select the features, how to make efficient indexing and high performance precisely. However, as the issue of image privacy is getting more and more attention, some users prefer not to reveal the content of their

query image when the features are retrieved, not even to the database. Various privacy-preserving CBIR systems have been proposed, as we will discuss in Section 11.2.

11.1.2 Scalar Invariant Feature Transform

Scalar Invariant Feature Transform (SIFT) is one of the most popular image descriptors in CBIR to detect and describe image features [222, 221]. The original SIFT descriptor was computed from the image intensities around interesting locations in the image domain which can be referred to as interest points, or key points. The image descriptor is computed at each interest point. The original SIFT descriptor proposed can be considered as a position-dependent histogram of local gradient directions surrounding the interest point. The SIFT descriptor is invariant to translations, scale and rotations transformations in the image domain. To achieve scale invariance of the descriptor, the size of location-based neighbours requires to be normalised in a scale-invariant way. To achieve rotational invariance of the descriptor, a dominant orientation in the neighbourhood is decided based on the direction of the gradient vectors.
The key points of SIFT can be briefly summarized as follows. Firstly, the size of the area surrounding the interest point is estimated by the detection scale of the interest point times a constant. Secondly, the local histogram of gradient directions is calculated by accumulating over the neighbour surrounding the interest point, therefore a preferred orientation estimate for the interest point can be decided. Thirdly, after the estimate of orientation and scale for the interest point is obtained, a rectangle grid with the interest point as center will be set. The orientation is decided by the main peaks of the histogram, and the size is the detection scale of the interest point times a constant. Then, after the vectors of image descriptors are calculated from two images, they are matched with each other by locating the points which minimize the distance (e.g., Euclidean distance) of the vectors. Finally, the descriptor is normalized.
There are a few extensions for SIFT. Dense SIFT was proposed by Bosch et al [72, 73], which is often used in object category classification. In object category classification, better classification results could be obtained by calculating SIFT descriptor over dense interest points. Other SIFT descriptors extending SIFT from grey-level to color images [72, 303, 78, 26]. PCA SIFT is another method for local image descriptors [193]. Similarly to regular SIFT, PCA SIFT detects interest points with scale estimates. However, the underlying image measurements are different. They calculate local maps of the gradient magnitude, instead of gradient orientations.
The SIFT descriptor and its extensions has been proven to be very useful in many applications, including object recognition, visual search [125], object category classification [73, 242], and so on.

11.1.3 Speeded Up Robust Features

Speeded Up Robust Features (SURF) [57] is similar to SIFT descriptors in the sense that it is also a feature vector derived from receptive-field-like responses in a neighbourhood of an interest point. However, there are several differences between SURF and SIFT: approximations of scale-space extrema; SURF is based on Haar wavelets instead of derivative approximations; the components of feature vector are computed as sums and absolute sums of first-order derivatives instead of histograms of gradient directions.

11.1.4 Shape-based Image Features

As mentioned in Section 11.1.1, shape is one of the key visual features for distinguishing an image. In shape-based feature extraction, shapes (line segments, circles, etc) are collected as a set of features for the images. One widely used shape-based feature extraction is using Hough Transform method [182]. Other shape-based systems use measures of moment invariants and Fourier descriptors, especially in 2D shape searching [151, 232].

11.2 Privacy-Preserving Techniques for Image Processing

Image processing operations can be delegated to clouds to ease computational overhead on the client side. To prevent the leakage of private information of the image owner or users who are querying, various privacy-preserving techniques for image processing have been investigated. Even features extracted from the image may reveal important private information of the owner/user. The attacker can deduce the image content by comparing features in a benchmark image database, or even recover part of the image based on these features. Therefore, we discuss existing popular privacy-preserving techniques for the above image processing algorithms accordingly.

11.2.1 Content Based Image Retrieval

Do et al. study the security issues of the typical technology blocks used in content-based image retrieval system [132]. There are work focusing on the functionality of image retrieval in the encryption domain. Some works target image feature protection where the similarity measures can be operated in the encryption domain. Lu et al. propose to encrypt images and their features separately [224]. Zhang et al. propose a solution to allow histogram-base image retrievals when the images are encrypted by permuting DCT coefficients [320]. Homomorphic encryption is another solution of retrieving and processing image content [224, 321, 319]. By combining signal processing and cryptographic techniques, Lu et al. propose three schemes where the similarity can be compared among protected features [223]. Bellafqira et al. propose a secure implementation of CBIR in homomorphic encryption domain that makes possible diagnosis aid systems to work in externalized environment [60]. Erkin et al. propose a privacy-preserving biometric face recognition protocol based on additive homomorphic encryption where a query image is sent homomorphically encrypted to the server [145]. Avidan et al. propose a so-called Blind Vision scheme which uses secure multi-party computation to vision algorithms [45]. Some research targets performing certain computation/processing over the encrypted data based on Shamir's Secret Sharing. These works assume that the cloud providers are trustworthy and therefore will not reveal private information. Lathey et al. propose a scheme which applies low-pass filtering to the ciphertext [213]. Later, they also propose a few image enhancement operations to the encrypted image data [214]. Upmanyu et al. extract image frames from video stream and detect the difference, to obtain the goal of video surveillance [302]. Another encryption direction for image protection is block-based image encryption algorithms. Besides general solutions like multi-party protocol and homomorphic encryption, there are also tailor-made solutions, which could be more efficient. Shashank et al. propose a private CBIR where the database is indexed using hierarchical index structure or hash based indexing scheme [283]. Their scheme can retrivie similar images from an image database without revealing the query image, while the image database is not encrypted. Weng et al. propose a privacy protection framework for large-scale CBIR [315], providing two levels of protection, where the privacy protection level can be adjusted according to a policy. Bernardo et al. propose a scheme called IES-CBIR for image data [146]. In IES-CBIR, both the data storage and query image are encrypted. Chou et al. propoase a block-based transformation algorithm for the purpose of image content protection, while the operations of image retrieval and image convolution can still be performed on the content-protected images [109].

11.2.2 Scalar Invariant Feature Transform

In this section we focus on secure image feature extraction algorithms using a *Scalar Invariant Feature Transform (SIFT)* descriptor. For secure applications of SIFT, Roy et al. propose to generate hash sequences from thresholding SIFT feature vectors [272]. Hsu et al. point out that SIFT features can be deleted or destroyed while maintaining acceptable visual qualities. Based on this, they propose an improved scheme to enhance the security of SIFT by introducing a key-based transform process to images [179]. Hsu et al. also propose a homomorphic encryption-based privacy-preserving SIFT method for feature extraction, based on Paillier cryptosystem [180]. Later, a privacy-preserving

implementation of SIFT has been proposed by using the Paillier's homomorphic cryptosystem [178] and improved by order-preserving encryption (OPE) [265]. Qin et al. propose a privacy-preserving SIFT feature detection system called SecSIFT [265], which distributes the computation procedures of SIFT to a set of independent, co-operative cloud servers. Schneider et al. further study Hsu et al.'s scheme, proposing optimization and shortcomings of their secure comparison protocol [27]. Schneider et al. improve the scheme by shifting computation from the server to the user. They also propose to use interactive comparison protocols or non-interactive somewhat or fully homomorphic encryption as alternatives.

11.2.3 Speeded Up Robust Features

Speeded Up Robust Features (SURF) is an enhanced version of SIFT. Bai et al. propose to use the Paillier cryptosystem to construct a multi-round interactive protocol to enable the cloud server to perform SURF on encrypted images [46]. Bai et al.'s scheme conduct the operations of interest point locations and feature descriptor extraction on the encrypted data. In their experiments, the location and number of interest points in the encrypted image and the same as those from the original image. Aiming to reduce the communication overhead of Bai et al.'s scheme, Wang et al. propose another privacy-preserving outsourcing scheme for SURF which preserves its key characteristics in terms of distinctiveness and robustness [311]. They split the original image data randomly and distribute the encrypted data shares to two independent cloud servers, in order to construct two new efficient protocols for secure multiplication and comparison. Somewhat homomorphic encryption (SHE) and single-instruction multiple-data (SIMD) are the two main techniques in their design.

11.2.4 Shape-based Image Features

Wang et al. present two schemes for shape-based image feature extraction by combining Hough transform, Pailliers homomorphic encryption, Gaussian Blur, and Garbled Circuits [312].

11.2.5 Techniques in other Applications

In the area of secure image processing, related work also includes the manipulation of encrypted image data in different applications, e.g., face recognition [193, 274, 252], fingercode authentication [53], ECG signals [52], and so on. Erkin et al. firstly propose a privacy-enhanced face recognition scheme which can hide both the biometrics and the result from the server that performs the matching operation efficiently [145]. In Erkin et al.'s scheme, they use secure multi-party computation. Sadeghi et al. propose a privacy-preserving face recognition scheme which substantially improves the communication and computation efficiency compared with Erkin et al.'s scheme[274]. Erkin et al.'s scheme requires O(log M) rounds and computationally expensive operations on homomorphically encrypted data to recognize a face in a database of M faces, while Sadeghi et al.'s scheme requires only O(1) rounds and has a substantially smaller online communication complexity (by a factor of 15 for each database entry) and less computation complexity. Sadeghi et al.'s scheme is based on garbled circuit. Osadchy et al. propose a so-called SCiFI system face identification using secure computation techniques [252]. Barni et al. propose a privacy-preserving protocol for fingerprint-based authentication using multi-party computation and homomorphic encryption [53]. Barni et al. present two methods for privacy-preserving ECG classification [52]: one is based on linear branching programs and the other one relies on neural networks.

11.3 Performance and Security Analysis

Given different descriptors in the image processing algorithms and their corresponding privacy-preserving implementation, the above mentioned schemes have different performance and security. Hsu et al.'s scheme [178] has a high computational complexity on encrypted data comparison. Their scheme uses

homomorphic encryption scheme (Paillier) and requires the cloud to traverse half of the ciphertext space for each comparison. In Hsu et al.'s scheme, the data owner needs to encrypt and decrypt each pixel in the image under Paillier scheme, which a huge overload for data owner. Qin et al.'s scheme [265] distributes the computation of SIFT to independent but co-operative cloud servers, therefore the outsourced computation can be kept simpler compared with Hsu et al.'s scheme. Two anti-SIFT attacks proposed by Hsu et al. can remove interest points retrieved by traditional SIFT [179]. Subsequently, Hsu et al. perform the security analysis under the attack models of ciphertext only and known plaintext for the proposed scheme. Wang et al. evaluate their secure SURF scheme regarding effectiveness and efficiency [311], and compare their scheme with the original SURF [57] and Bai et al.'s scheme [46]. For efficiency, they evaluate the communication and time cost on the data owner with different size of images. Compared with Bai et al.'s scheme, Wang et al.'s scheme reduce the computation burden on the data owner. Moreover, in the evaluation results, Wang et al.'s scheme has an even lower than doing SURF locally especially for large images. By combining homomorphic encryption and garbled circuit, Sadeghi et al.'s scheme [274] improves the communication and round complexity compared with Erkin et al.'s scheme, under different security parameters. In general, homomorphic encryption based algorithms result in heavier workload compared with tailor-made solutions.

11.4 Efficient Implementation of Image Processing in SGX

In this section we propose an approach for implementing efficient privacy-preserving image processing with the help of hardware features provided by Intel Software Guard Extensions (SGX). SGX is an emerging technology providing improved isolation capabilities for computations, which can be used protect sensitive computations and data.

11.4.1 Software Guard Extensions (SGX)

SGX is an extension of the x86-64 instruction set architecture that allows to instantiate trusted execution environments, called *enclaves*. SGX enclaves are isolated from the rest of the system. In contrast to other trusted execution environments [42] SGX does not require trust in any hardware but the CPU. Trust in the CPU is necessary because the access control to enclaves, i.e., enclave memory, is enforced within the CPU.
The application developer can divide her application into an *untrusted* and *trusted* part which should be executed in enclave mode. To start an enclave, the applications defines a region of memory which should be used for the enclave using the newly introduced SGX instructions. Once the enclave is initialized the memory region that was assigned to the enclave becomes inaccessible to the rest of the system. This includes higher privileged modes such as kernel, hypervisor, and system management mode (SMM). In fact, the memory can only be accessed by the code of the enclave. As mentioned before, SGX does not require any trust in the underlaying hardware. This is implemented by encrypting and integrity protecting enclave memory with enclave-specific keys before the memory physically leaves the CPU cache and is written to the external memory (RAM). If an enclave wants to access enclave memory that was swapped into RAM, the CPU will fetch the data, check the integrity, and decrypt it only after the integrity has been verified. This ensures that an attacker cannot tamper with enclave memory that was stored outside the CPU. Further enclaves ensure that when the CPU changes from enclave mode to non-enclave mode, that all registers are securely stored in memory, so they cannot be modified, and cleared, so no information from inside the enclave are leaked.
After an enclave is started, the untrusted part of the application can communicate with the trusted part of the application (which runs in enclave mode) by calling functions that were defined during enclave initialization. These well defined entry points prevent attackers to execute enclave code starting from the middle of a function or even an instruction which could result in unwanted behavior. On the other hand, the trusted code of the enclave can access the whole application memory.
To ensure that the trusted part of an application was not modified before it is loaded into the enclave, SGX features attestation. Specifically, the CPU creates a hash value over the whole enclave memory

before initializing it. This hash can then later be used to attest that the trusted part of an application was not modified before or during the creation of an enclave. Further, an enclave can attest that it is executed in a benign SGX enclave, and not in an emulated environment.

11.4.2 Overview of SGX-based Privacy-Preserving Image Processing

To achieve both security and efficiency, SGX can be used to load algorithm code and data into enclave. Existing solutions for Bajaj et al. propose a trusted hardware-based database called TrustedDB [48]. Baumann et al. propose Haven using instruction-level isolation mechanism of SGX to protect malicious host OS [56]. However, existing isolating databases with SGX load a whole unmodified database management system into an enclave without considering information leakage. We take into account the side channel information which could be leaked between trusted environment and untrusted host. Moreover, an enclave has a limited memory size, which could be not very feasible to load the whole database system.
Leveraging Intel's SGX, we present a hardware-based solution which is suitable for achieving privacy-preserving of image processing algorithms while still guarantee high efficiency. We provide two SGX constructions, dependent on how to manage the B+ tree of database management systems. In the first construction, the whole database will be loaded and processed inside the enclave. The client constructs the B+ tree locally, encrypts the structure of B+ tree and then sends them to the cloud. The SGX application is deployed to a SGX capable server. The second construction only load the candidate nodes for the tree traversing, instead of the whole database. One advantage of this construction is that it can reduce the requirement of memory inside the enclave to O(1) for the tree. All nodes are encrypted in the regular memory or disk, and will be decrypted by the enclave. The challenges of this construction are a slightly larger leakage compared with the first construction. In general, our proposed solutions are very efficient. It has a logarithmic complexity in the size of database. The searches can be performed in a few milliseconds. Besides, we formally prove the security of our scheme. Our implementation also has a small code and memory footprint.

11.4.3 Structure of SGX-based Privacy-Preserving Image Processing

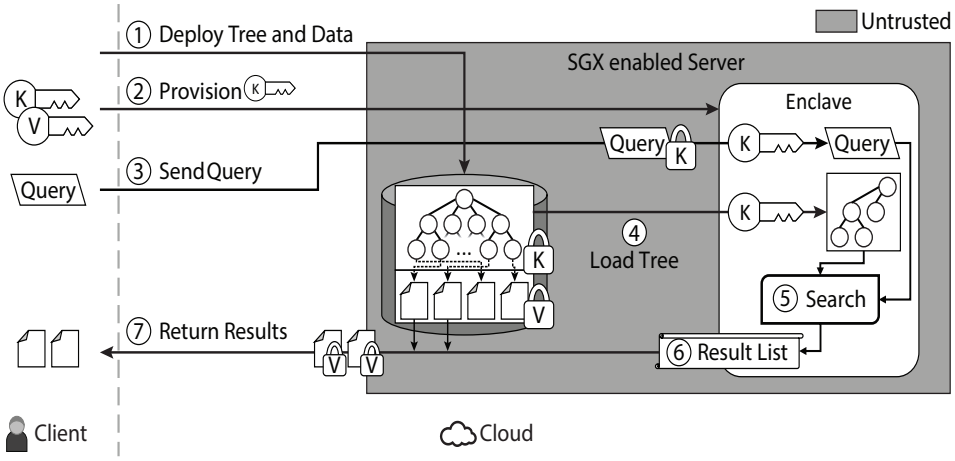


Figure 11.1: Structure in high level

As shown in Figure 11.1, our scheme contains three parties: an SGX enabled server (which is untrusted), the hardware protected SGX enclave within the server and the client (the data owner who is trusted). Our scheme includes several steps. Firstly, the client augments the data values with search keys. The keys and the corresponding values are inserted into a B+ tree. This value can be data in a relational database or files. Multiple search keys can be supported by different trees. The data nodes

and the values are stored in a pseudo-random order. The tree and values are linked via pointers which are added to the leaves of the tree and can identify the position of the related values.
All nodes in the tree are then encrypted by the client with a secret key SK_k. All data values are encrypted by the client with a secret key SK_v. The encrypted B+ tree and the encrypted values are then deployed on the server in the cloud which is untrusted (Step 1 in Figure 11.1). The tree nodes and values can be encrypted as a block or individually, depending on the visualizing setting or real setting. In Step 2, a secure link is built between the enclave and the client. The client utilizes the attestation feature of SGX to authenticate the enclave and provisions SK_k into the enclave via this secure connection (Step 2). The one time setup of our scheme is completed by this step.
Afterwards, the client can send search queries of image processing to the server. All search queries which are encrypted by SK_k is under randomized encryption. Therefore, the untrusted server is not able to learn information about the image query, even if the same query has already been sent before. When the image query arrives in the enclave, SK_k can be used to decrypt this query (Step 3).
The enclave then loads the structure of B+ tree (only including tree nodes but not values) into the memory of enclave from the untrusted storage, and decrypts the B+ tree (Step 4). When the memory is sufficient, the entire B+ tree will be loaded into the enclave. Then the search can be performed (Step 5).
However, in a more often case, the size of the B+ tree can be much bigger than the size of the available memory within the enclave, therefore we provide another design which is suitable for this case: only a subset of the tree nodes will be loaded into the enclave. After this, we traverse the B+ tree from the root node. When the image query arrives to a node which does not exist in the enclave at that moment, this node will be fetched from the untrusted storage.
In Step 6, the queried results which are represented by a list of pointers are then returned to the untrusted part. Note that during this interaction the untrusted part cannot learn anything, except for the cardinality of the result set, since the values are stored in a randomized order. In both cases we mentioned above, the search algorithm will always reach to a set of nodes, which holds a list of pointers to data values which match the query.
In the last step, with the returned pointers, the server fetches the encrypted image values from the untrusted storage and passes these values to the client (Step 7). The client then decrypts the received values and conducts further image processing.
In both proposed schemes, the server never receives the image data in plaintext. We can encrypt these data with any strong standard cryptographic encryption algorithms (e.g., AES-128 in GCM mode). The server is also never able to decrypt the data, even within the SGX enclave. Only the client possesses the key to decrypt the image data

11.4.4 Adversarial model and related assumptions

In this section we highlight some assumptions in our scheme and the adversarial model. We have three main assumptions in our scheme. Firstly, we assume that a system which provides SGX (or any system having a TEE with capabilities similar to SGX). Secondly, we assume that code and data within the TEE are protected regarding their integrity and confidentiality. Lastly, we assume that the TEE provides methods to build a secure channel to the client which allows secure communication as well as secure provisioning.
Our adversarial model is shown in Figure 11.2. The attacker cannot access the enclave directly due to the protection of SGX. However, the attacker might potentially achieve sensitive information through side channels. The goal of the attack is to learn the structure of the B+ tree which shows him the relation between the stored data. If the attacker is able to learn the structure of the B+ tree, he can further use this information to derive other information about the data. At the beginning, the structure of the tree is hidden from the attacker, since the tree nodes are stored in a randomized order on the server.
Our scheme resists the following attacker (Figure 11.2):

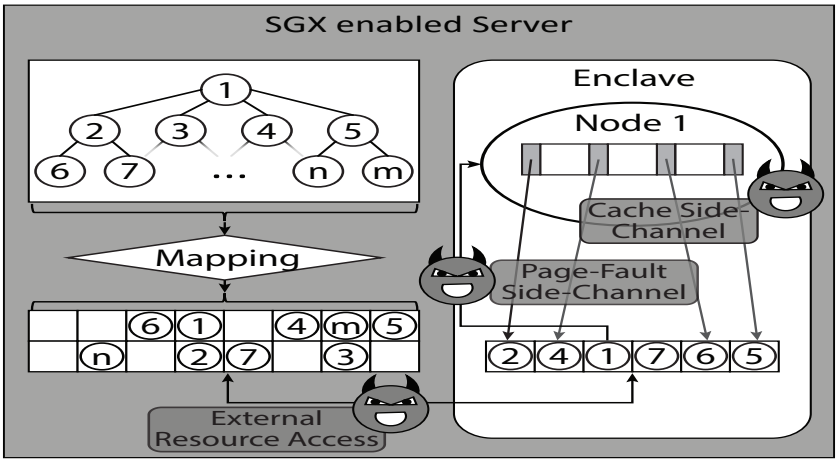


Figure 11.2: Adversary channels

1 The attacker can use page-fault side channel to observe data access inside the enclave at page granularity, and use this side channel to observe access patterns on the B+ tree stored within the enclave.

2 The attacker can observe all interaction of the enclave with resources outside the enclave. In particular, the attacker can observe the access pattern to B+ tree nodes stored outside the enclave.

3 The attacker can use cache side channel to learn about code paths or data access patterns inside the enclave, as SGX does not provide protection. In this case, third party libraries, like cryptographic libraries, are assumed to be secure against cache side channel.

Our scheme doesn't consider hardware attacks and Denial of Service attacks.

Chapter 12 Other Encryption-based privacy-preserving components

In this chapter, we focus on encryption techniques used to protect data in the SUPERCLOUD infrastructure. Our solutions are mainly compatible with any kind of storage (blocks, objects, files), even if some of these specifications are given in the case of a file system (see Section 12.3). In this chapter, we also focus on the data deduplication problem and give a concrete solution.

12.1 Context

Cloud data storage is a widespread option and it is today easy to find such solutions for any digital content. A key issue is the confidentiality of stored data, which becomes even trickier when the owner wants to share her/his data. It is then crucial to offer solutions that combine storage, sharing and confidentiality of data, without changing the user experience.
The idea behind cloud storage is that data are stored as if they were in a safe, where the cloud storage plays the role of an access control to this safe. In reality, the data are encrypted but, most of the time, the cloud server has the decryption key and manages the rights for each user to access or not the data.
This is a critical problem in the case of private sensitive data such as administrative documents (*e.g.* identity cards, bills or pay sheets) or, as in the use cases described in the SUPERCLOUD project, *health data*. For instance, the result of a medical exam is typically accessible to all medical doctors, but not to other types of medical professionals (e.g., IT or administrative). The results of more private exams may be only accessible by specific medical doctors. Additionally, the Cloud Service Provider storing the data is obviously not allowed to obtain information about these exams. This is also a tricky point in the case of confidential documents owned by a business enterprise and shared between collaborators or with trading partners.
The main problem is that simple encryption only provides a backup service, and not a secure storage one with practical features. More precisely, it does not work when one wants to share the stored data with other customers. In the SUPERCLOUD project, we solve this problem by using advanced cryptographic tools:

- *proxy re-encryption*, with which it becomes feasible to share encrypted data with other users, known by their identities. For example, the exam's lab can encrypt the medical exam before sending it to the CSP, and then share it with either all medical doctors (but not, e.g., IT professionals) or with some specific ones;
- *attribute-based encryption*, for which the data is encrypted according to a specific policy. In this case, only receivers with attributes satisfying this specific policy can decrypt the encrypted message. For example, the exam's lab can encrypt the medical exam with a specific access control policy *"any user with attribute medical doctor"*.

In this chapter, we specify both these solutions.

12.1.1 Some Notations

All along this chapter, a public key encryption scheme is given by the following algorithms: PKKeyGen is the key generation to create public pk and private sk keys, PKEnc is the encryption process and PKDec is the decryption.
The AES symmetric key encryption scheme is also used several times. Its description should be taken from the FIPS 197 on "Advanced Encryption Standard (AES)". Below, the AES encryption algorithm is denoted AESEnc and the decryption one is denoted AESDec.

12.2 Key Encapsulation and Deduplication

As shown in the context section above, we sometimes have to treat data which size can be big (e.g., medical analysis,). This makes public key cryptography unsuitable and poses some problem. To deal with that point, one may use the so-called "encapsulation technique", described in this section. Note that this one can be used in addition to the other public key encryption techniques describe later in this chapter.
We also focus on the deduplication problem, since it also refers to a method to efficiently encrypt the data itself, with some additional properties.

12.2.1 Key Encapsulation

Motivation. Public key cryptography better suits the addition of features on encrypted data. But the price to pay is that it is only efficient when the data to be encrypted are relatively small. In this section, we focus on this particular problem by explaining the way to perform key encapsulation, and then encrypt the data using much more performing secret key cryptography for the data itself.

Description. The encapsulation technique consists in encrypting the data using a symmetric encryption algorithm (say AES) using a secret key k that is next encrypted using an asymmetric encryption algorithm. Such procedure is given by Algorithm 11 for a data d.

Algorithm 11: Key encapsulation.

Input : data d, public key pk.
Output: encrypted data C_d and metadata MD_d.

1 generate e.g., an AES key k_d;
2 encrypt the data f using the AES algorithm and the key k_d as $C_d = \mathsf{AESENC}(d, \mathsf{k}_d)$;
3 encrypt the secret key k_d using the public key encryption procedure $\mathsf{MD}_d = \mathsf{PKEnc}(\mathsf{pk}, \mathsf{k}_d)$;
4 output (C_d, MD_d).

The ciphertext C_f is then the true encrypted file, while MD_f corresponds to some meta-data associated to the encrypted file. We consider that these meta-data also contain the identity of the client uploading the file.

Remark 1. *In the first part of this section, dedicated to privacy preserving techniques, we will focus on these three steps and the way one can implement them for data confidentiality. More precisely, the first and second steps can be implemented either by using standard techniques (e.g., AES), or using the convergent encryption [137] that is given in the next section. Then, in the third step, the used public key encryption procedure* PKEnc *can be implemented either by a standard encryption algorithm such as RSA [270] or ElGamal [154], by a proxy re-encryption scheme if one wants to share the stored data (see Section 12.3) or by an attribute based encryption if one wants to give an access control policy to stored data (see Section 12.4).*

12.2.2 Convergent Encryption for Deduplication

Motivation. Cloud computing is often promoted towards companies as a way to reduce their costs while increasing accessibility and flexibility. It is common sense to have one large computing infrastructure that companies would share instead of replicating smaller ones. This saves money and is an eco-friendlier way to distribute resources. But cloud platforms are neither cheap nor eco-friendly. The larger amount of data these platforms host, the more expensive they become. Impact on the environment grows as well. One way to address this issue is to delete identical files stored by users. This method, called *deduplication*, is widely used by cloud providers. However, some of the cloud storage users may want to encrypt their data, distrusting honest-but-curious providers. If they use a classical encryption scheme, deduplication is not possible anymore: two encryptions of the same plaintext under different keys yield indistinguishable ciphertexts. A new kind of encryption is needed, under which it is possible to determine whether two different ciphertexts are *locked* to the same message or not. In this document, we describe the *convergent encryption*.

Description. The convergent encryption [137] works as follows, where $\mathcal{H}$ is a hash function and AES is the deterministic version of the standard symmetric encryption system.

- $\mathsf{CESetup}(\kappa)$ returns the global parameters param.
- $\mathsf{CEKeyGen}(\mathsf{param}, m)$ returns $\mathsf{k}_{ce} = \mathcal{H}(m)$.
- $\mathsf{CEEnc}(m, \mathsf{k}_{ce})$ returns the ciphertext $C = \mathsf{AESENC}(m, \mathsf{k}_{ce})$.
- $\mathsf{CEDec}(C, \mathsf{k}_m)$ gets back the plaintext $m = \mathsf{AESDEC}(C, \mathsf{k}_{ce})$.
- $\mathsf{CETest}(C_1, C_2)$ returns 1 iff $\mathcal{H}(\mathsf{param}\|C_1) = \mathcal{H}(\mathsf{param}\|C_2)$. The value $\mathcal{H}(\mathsf{param}\|C)$ is called a "tag".

Remark 2. *As explained in [61], this convergent encryption guarantees all desired security properties and has also good properties regarding performances. Its main drawback is that the time to compute a tag (i.e., $T = \mathcal{H}(\mathsf{param}\|C_1)$) can be considered as too long. One may thus prefer to use other close constructions, such as HCE2 or RCE (see also [61] for a description of those schemes). The counterpart is that the encryption process is less efficient.*

Usage. If one wants to implement data deduplication with key encapsulation, one has to modify the first and second steps above and replace them by respectively the $\mathsf{CEKeyGen}$ and the CEEnc given by the convergent encryption scheme. The Algorithm 12 gives the way to execute the key encapsulation with the convergent encryption.

Algorithm 12: Key encapsulation with convergent encryption.

Input : data d, public key pk.
Output: encrypted data C_d and metadata MD_d.

1 executes the $\mathsf{CEKeyGen}$ procedure of the convergent encryption to obtain the key k_{ce};
2 encrypt the data f using the convergent encryption procedure CEEnc and the key k_{ce} as $C_{ce} = \mathsf{CEENC}(d, \mathsf{k}_{ce})$;
3 encrypt the secret key k_{ce} using the public key encryption procedure $\mathsf{MD}_d = \mathsf{PKEnc}(\mathsf{pk}, \mathsf{k}_{ce})$;
4 output (C_d, MD_d).

12.2.3 Encrypted Data Storage Sequence Diagrams

We consider a user, say Alice, wanting to upload and then download a file in an encrypted manner to a Cloud Service Provider. For that purpose, Alice makes use of the SUPERCLOUD Proxy and

Server. We here consider that the PKSetup and PKKeyGen procedures for a public key encryption scheme have already been executed.

File upload. After the validation of the user (steps 1 to 6), this step, given in Figure 12.1, executes the key encapsulation procedure (steps 7 to 9, see Algorithm 11 or 12). The algorithm outputs the encrypted file C_f sent to the Cloud Service Provider (steps 11 and 12) and the meta-data MD_f sent to the WP3 SUPERCLOUD server (steps 13 and 14).

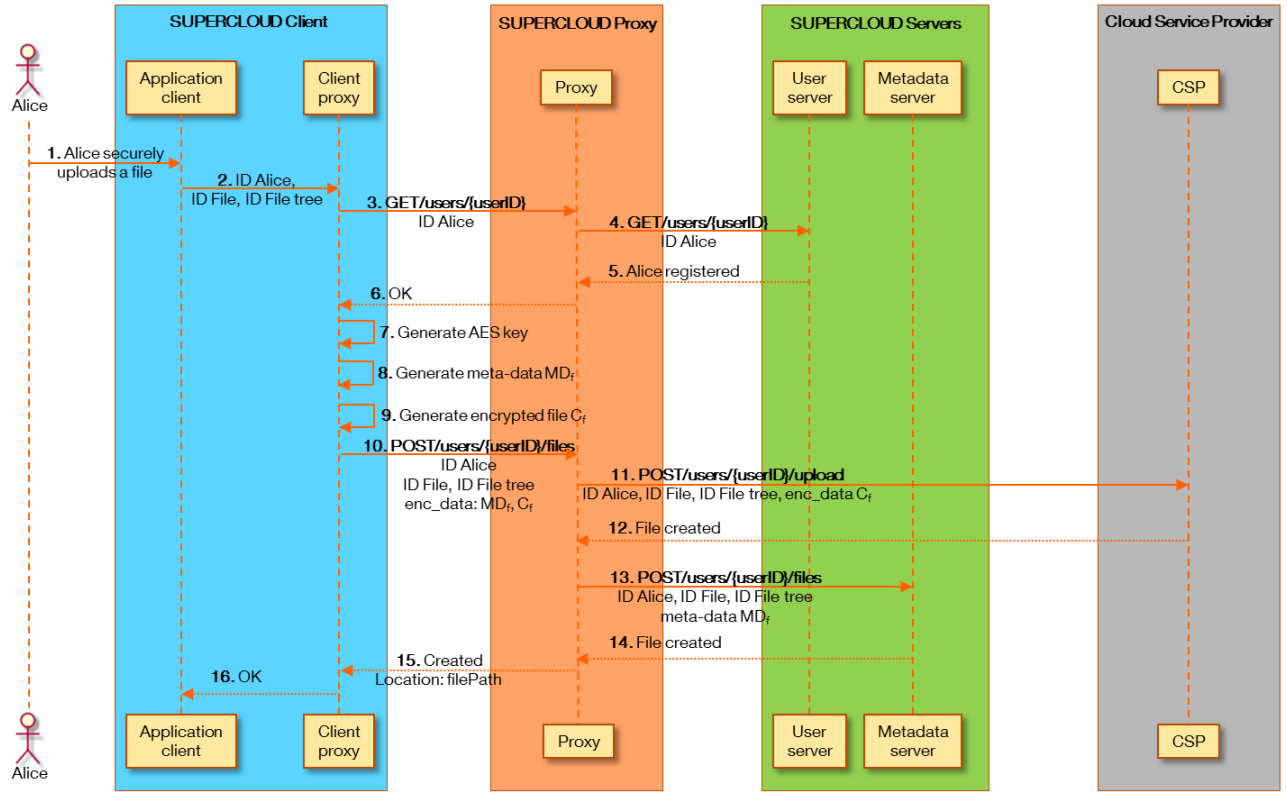


Figure 12.1: Upload phase

File download. As shown in Figure 12.2, the client gets back the meta-data MD_f from the SUPERCLOUD server (steps 4 and 5), and the encrypted file itself is simply requested from the corresponding Cloud Service Provider (steps 6 and 7). Then, the client can use its secret key to decrypt the meta-data MD_f to obtain the AES key k_f (step 9), and then use the AES decryption algorithm with k_f and C_f to obtain the file f in clear (step 10).

Data deduplication. The implementation of a data deduplication mechanism in this system necessitates an additional procedure to test whether a new uploaded file is a duplicate or not. Figure 12.3 gives the modifications that should be done to the above downloading phase (given by Figure 12.2).

Remark 3. *One may notice that a specificity of convergent encryption is that the way to obtain the file f in plain necessitates the use of the key k_f, which one is derived from the file f. This may be seen as contradictory, but the use of the above encapsulation method permits to easily solve that point.*

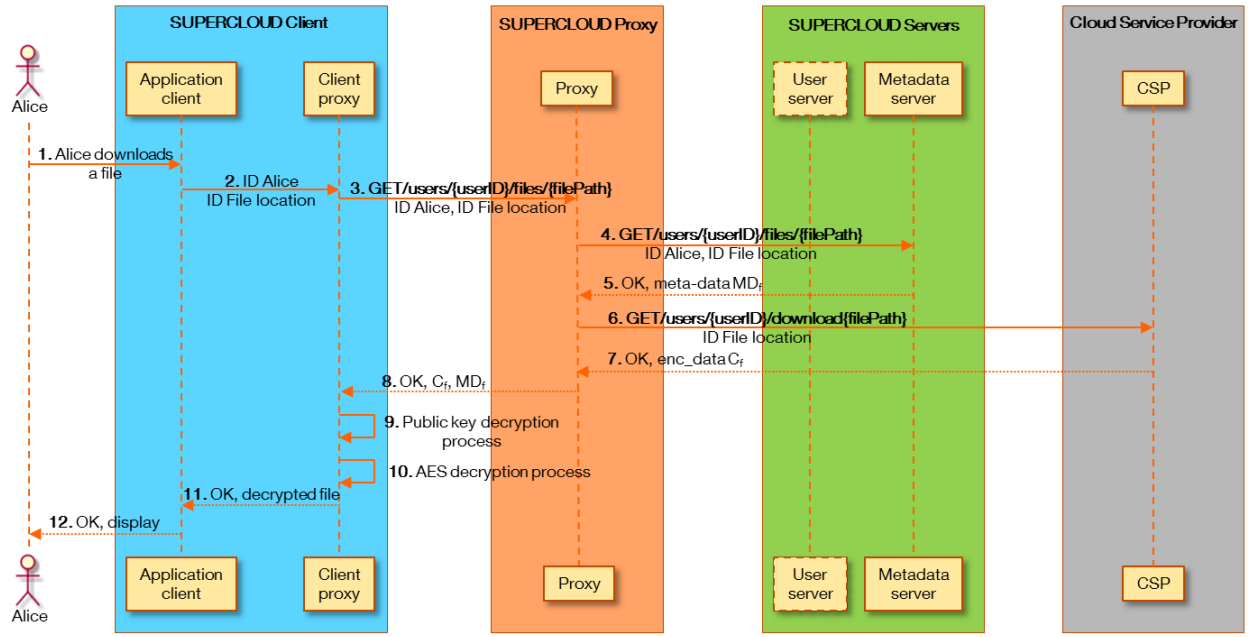


Figure 12.2: Download phase

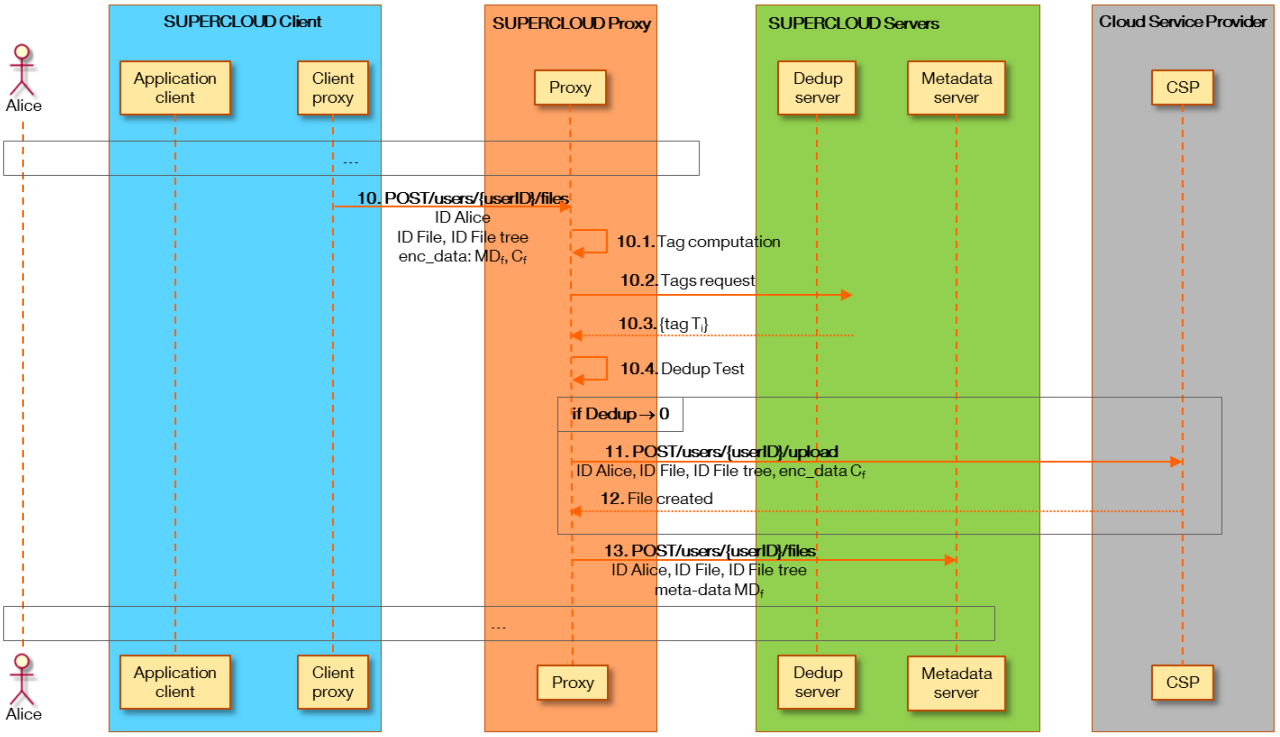


Figure 12.3: Additional interactions in the download phase for deduplication

12.3 Proxy re-encryption

In deliverable D3.1, we have shown that proxy re-encryption is a cryptographic tool that can be used to protect data, while permitting clients to share it securely. In this deliverable, we focus on the way such tool can be used to store and share data within a file system, inside the SUPERCLOUD WP3

architecture.

12.3.1 Proxy re-encryption in a nutshell

Motivation. We consider a set of users wanting to store and share some documents to a Cloud Service Provider. As they do not trust the CSP, they first encrypt their data before sending it. We here focus on the sharing procedure. At first, Alice uploads a new document, in an encrypted form, to the CSP. then, she wants to share such document to say Bob, in such a way that the CSP does not learn any information about the data. Finally, Bob downloads Alice's document.
Description. In a nutshell, a Proxy Re-Encryption scheme (PRE for short) allows a user to delegate its decryption capability in case of unavailability. To do so, this user, Alice, computes a *re-encryption key* $\mathsf{rk}_{A\to B}$ which is given to a proxy. The key $\mathsf{rk}_{A\to B}$ allows the proxy to transform a ciphertext intended to Alice into one intended to Bob. While doing this, the proxy does not learn *any* information on the plaintexts nor any secret key. In the sequel, we focus on (1) *unidirectional* and (2) *single-hop* PRE schemes, which means (1) that with a re-encryption key $\mathsf{rk}_{A\to B}$, a proxy cannot translate Bob's ciphertexts into ciphertexts intended to Alice and (2) that once a message has been moved into a ciphertext intended to Bob, no more transformation on the new ciphertext intended to Bob is possible. Such cryptographic scheme is composed of seven algorithms that have been detailed in D3.1. We here only recall the main characteristics and inputs/outputs. We recall that κ is a security parameter giving the size of the parameters and keys that have to be used, and $\perp$ denotes an error message. The parameters param are given on input to all algorithms but, for sake of clarity, we omit them in the description.

- $\mathsf{Setup}(\kappa) \to \mathsf{param}$.
- $\mathsf{KeyGen}(\mathsf{param}) \to (\mathsf{sk}_A, \mathsf{pk}_A)$.
- $\mathsf{ReKeyGen}(\mathsf{sk}_A, \mathsf{pk}_B) \to R_{A\to B}$.
- $\mathsf{Enc}(\mathsf{pk}_A, m) \to C$.
- $\mathsf{ReEnc}(\mathsf{rk}_{A\to B}, C) \to C' / \perp$.
- $\mathsf{Dec}_1(\mathsf{sk}_B, C) \to m / \perp$.
- $\mathsf{Dec}_2(\mathsf{sk}_A, C) \to m / \perp$.

Within the SUPERCLOUD WP3 architecture, as explained in D3.1, the WP3 proxy is responsible for the ReKeyGen procedure and the WP3 servers manages the storage of (i) re-encryption keys and (ii) file meta-data.
We first describe the cryptographic algorithm that is at the core of the implemented PRE scheme. We then do not give all the mathematical details of the used PRE scheme, but only an overview of the scheme. We also give the main interactions between the client, the proxy and the server, when a file is uploaded, shared and finally downloaded.

12.3.2 Cryptographic Basis

A close look on related work shows that unidirectional single-hop schemes are mainly based on the seminal work by Blaze, Bleumer and Strauss [69], itself based on the ElGamal encryption scheme [154]. For efficiency reasons, we focus on the hash ElGamal encryption scheme (in its IND-CPA version [104]), which is now given.

Hash ElGamal scheme. Let $\mathbb{G} = \langle g \rangle$ be a group of prime order p with a multiplicative law, as in the original paper, and let $\mathcal{H}$ be a collision-resistant hash function. The symbol $\oplus$ denotes the bit-wise or operation.

- Private key $x \in_R \mathbb{Z}_p^*$ and public key $X = g^x$.
- Encryption[1]: $T_1 = m \oplus \mathcal{H}(g^r)$ and $T_2 = X^r$ where $r \in_R \mathbb{Z}_p^*$.
- Decryption: $m = T_1 \oplus \mathcal{H}(T_2^{1/x})$.

Proxy Re-Encryption. Based on (hash) ElGamal, Blaze *et al.* [69] have proposed a bidirectional multi-hop PRE scheme that can then be transformed into a secure unidirectional and single-hop PRE scheme, using the technique given in [110, 90]. We do not give the details and refer the reader to the corresponding references.

Conditional PRE. Finally, our construction makes use of a "conditional" PRE [298]. In such variant, the encryption process is related to a chosen condition γ_0, and the re-encryption key is generated under a condition γ_1. Given a decryption key and the condition γ_1, one can decrypt as usual. Moreover, if $\gamma_0 = \gamma_1$, the re-encryption process outputs a value that can be normally decrypted.
More formally, we have:

$$\begin{aligned} c &= \mathsf{Enc}(\mathsf{pk}_A, m, \gamma_0) \text{ and} \\ \mathsf{rk}_{i \to j, \gamma_1} &= \mathsf{ReKeyGen}(\mathsf{sk}_i, \mathsf{pk}_i, \mathsf{pk}_j, \gamma_1). \end{aligned}$$

If $\gamma_0 = \gamma_1$, then

$$\mathsf{Dec}_1(\mathsf{sk}_j, \mathsf{ReEnc}(\mathsf{rk}_{i \to j, \gamma_1}, c)) = m$$

and otherwise, the level 1 decryption process outputs $\perp$. A concrete construction can be found in [298, 89].

12.3.3 Management of a File System

Objective. A standard PRE scheme has an "all or nothing" sharing property. If the re-encryption key is generated by Alice, then the proxy can re-encrypt for Bob any document initially encrypted to Alice. There is no way for Alice to restrict what the proxy can re-encrypt or not, except by trusting it. But if the storage space is structured as a tree (as shown in Figure 12.4), Alice may want to only share a specific folder, or a specific files, but not all her files.

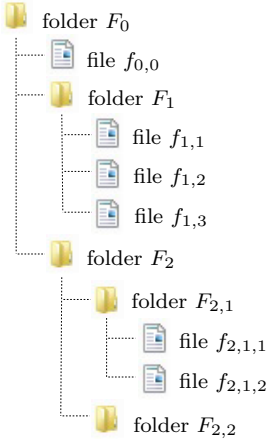


Figure 12.4: A tree structure

[1] One may note that the role of the generator g and the public key X are inverted *w.r.t.* the traditional ElGamal scheme. This does not compromise the security, and is necessary to obtain a PRE, as shown in [69].

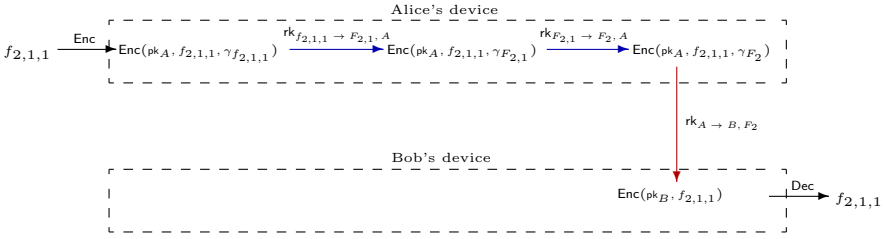


Figure 12.5: Re-conditioning principle

Main ideas. For this purpose, one solution is *conditional* PRE (see above). But this is not enough in our case. In fact, this can only be helpful to manage a "file by file" sharing, but not a true file system, since there is no possible link between a file and the folder to which it belongs. The idea, illustrated in Figure 12.5, is then to manage a two directional conditional PRE, one to go from one folder to another, and the other to go from one user to another.
We then attach to each uploaded file a unique condition, defined during the encryption process, and denoted $\gamma_{f_{2,1,1}}$ for file $f_{2,1,1}$ for example. We then obtain the ciphertext $\mathsf{Enc}(\mathsf{pk}_A, f_{2,1,1}, \gamma_{f_{2,1,1}})$, using Alice's public key pk_A. Knowing $\gamma_{f_{2,1,1}}$, Alice, the owner of the file, can easily decrypt it, using her own private key sk_A.
Then, if Alice wants to give the rights to *e.g.* Bob for folder F_2, she computes a re-encryption key, from Alice to Bob, under a condition related to F_2, denoted γ_{F_2}. Such re-encryption key is denoted $\mathsf{rk}_{A\to B,F_2}$ and is sent to the proxy to publish the validity of the sharing. It permits a vertical transformation between users, if and only if the conditions match. Going back to our example, this is not yet the case since the file is encrypted under condition $\gamma_{f_{2,1,1}}$, while the re-encrypted key has been computed under condition γ_{F_2}.
We then add an horizontal transformation inside the file system, using additional re-encryption keys such as $\mathsf{rk}_{f_{2,1,1}\to F_{2,1},A}$ or $\mathsf{rk}_{F_{2,1}\to F_2,A}$ (see Figure 12.5). These keys permit to go back up the tree from a file to a specific folder by modifying the condition attached to the ciphertext, while preserving the plaintext and the entity being able to decrypt the resulting ciphertext. As an example, using the re-encryption key $\mathsf{rk}_{f_{2,1,1}\to F_{2,1},A}$, one can transform the ciphertext $\mathsf{Enc}(\mathsf{pk}_A, f_{2,1,1}, \gamma_{f_{2,1,1}})$ into the ciphertext $\mathsf{Enc}(\mathsf{pk}_A, f_{2,1,1}, \gamma_{F_{2,1}})$. As shown in Figure 12.5, the re-encryption key $\mathsf{rk}_{F_{2,1}\to F_2,A}$ then permits to obtain $\mathsf{Enc}(\mathsf{pk}_A, f_{2,1,1}, \gamma_{F_2})$, that is the file $f_{2,1,1}$ encrypted under Alice's public key pk_A and condition γ_{F_2}.
Then, for each couple (file, folder) or (folder, folder) in the path from the file to the root, Alice needs to compute a re-encryption key (but only once for each link between folders).
Finally, when the encrypted file is related to a condition for which it exists a re-encrypted key $\mathsf{rk}_{A\to B,F_2}$ from Alice to Bob (in our example $\mathsf{Enc}(\mathsf{pk}_A, f_{2,1,1}, \gamma_{F_2})$), then the proxy can re-encrypt it into a ciphertext that can be decrypted by Bob, that is $\mathsf{Enc}(\mathsf{pk}_B, f_{2,1,1})$ (the condition is withdrawn during the mathematical transformation).

12.3.4 High Level Specifications

We now give a high level description of the scheme, which can be instantiated using the basic cryptographic scheme given above. The details can be found in [89].

Encryption process. During the encryption step, we use the conditional encryption algorithm, with a condition γ_f related to the uploaded/encrypted file f itself. We then obtain

$$c = \mathsf{Enc}(\mathsf{pk}, f, \gamma_f).$$

Tree re-encryption key generation. The novelty of this method is that the proprietary of the tree structure needs to generate a re-encryption key for each edge in the tree, *i.e.* each link folder-file or

folder-folder, as shown in Figure 12.4. Our example in this figure necessitates 10 re-encryption keys, such as $\mathsf{rk}_{A,f_{0,0}\to F_0}$, $\mathsf{rk}_{A,F_1\to F_0}$ or $\mathsf{rk}_{A,f_{2,1,1}\to F_{2,1}}$.
The main aim of these re-encryption keys will be to modify the condition related to the ciphertext so that it goes from the file to its belonging folder, then to the upper folder and so on until a given nonce in the tree structure. This should be done without modifying the underlying public key which has been used to encrypt the file. This will be the role of the user re-encryption key (the "vertical" one).
More precisely, given a ciphertext $c = \mathsf{Enc}(\mathsf{pk}, f, \gamma_f)$ (as computed above) and a tree re-encryption key $\mathsf{rk}_{A,f\to F}$ output by the $\mathsf{TReKeyGen}(\mathsf{sk}_A, \gamma_f, \gamma_F)$ algorithm, we describe a tree re-encryption procedure TReEnc such that

$$\mathsf{TReEnc}(\mathsf{rk}_{A,f\to F}, c) \quad = \quad \mathsf{Enc}(\mathsf{pk}, f, \gamma_F). \tag{12.1}$$

User re-encryption key generation. We consider that Alice wants to share a folder with Bob. For this purpose, Alice computes a re-encryption key as usual, using a condition dedicated to the file/folder she wants to share with Bob. For example, if Alice wants to share the folder F_2 with Bob, she has to compute

$$\mathsf{rk}_{A\to B,F_2} = \mathsf{UReKeyGen}(\mathsf{sk}_A, \mathsf{pk}_B, \gamma_{F_2}).$$

The main problem is now that it is necessary to modify the condition from the one related to the ciphertext, to the one related to the rights. This is precisely why such two directional PRE has been designed.

Re-encryption process. Let us suppose that Alice has computed a re-encryption key $\mathsf{rk}_{A\to B,F_2}$ related to folder F_2 for Bob. We also assume that Bob wants to access the file $f_{2,1,1}$. As explained above, this file is encrypted as $c = \mathsf{Enc}(\mathsf{pk}_A, f_{2,1,1}, \gamma_{f_{2,1,1}})$, for Alice.
If this is not already the case (as it is in our example), the first step of the re-encryption process

$$\mathsf{ReEnc}(\mathsf{rk}_{A\to B,F_2}, \{\mathsf{rk}_{A,f_{2,1,1}\to F_{2,1}}, \mathsf{rk}_{A,F_{2,1}\to F_2}\}, c)$$

consists in transforming c into a ciphertext c' such that

$$c' = \mathsf{Enc}(\mathsf{pk}_A, f_{2,1,1}, \gamma_{F_2}),$$

still for Alice. This is done by using the tree re-encryption keys $\mathsf{rk}_{A,f_{2,1,1}\to F_{2,1}}$ and $\mathsf{rk}_{A,F_{2,1}\to F_2}$, and the appropriate re-encryption algorithm, as described above in the tree re-encryption key generation paragraph, especially equation (12.1).
The second step is then the execution of the traditional re-encryption key algorithm, on input c' and $\mathsf{rk}_{A\to B,F_2}$ computed as shown above.
As the conditions match, both equals to γ_{F_2}, this re-encryption is now possible. The result is finally the ciphertext

$$c'' = \mathsf{Enc}(\mathsf{pk}_B, f_{2,1,1}).$$

Decryption process. This is obvious that Bob can decrypt the ciphertext c'' resulting from the whole re-encryption procedure to obtain the file $f_{2,1,1}$ in plain:

$$f_{2,1,1} = \mathsf{Dec}(\mathsf{sk}_B, c'').$$

12.3.5 Sequence Diagrams

In the previous section, we have described each algorithm composing our proxy re-encryption scheme. We now explain how those algorithms can be used in the WP3 architecture. For sake of simplicity, we consider that the Setup and KeyGen procedures have already been executed and we now focus on the file upload, file/folder sharing, and file download. In particular, we consider in the sequel that the tree re-encryption key generation given above has already been executed (except for the new uploaded file). We give the whole set of sequence diagrams for both upload and download, even if some parts

are quite close to the ones given previously in Figures 12.1 and 12.2.

File upload. This phase (see Figure 12.6) implies the execution of both the encryption process (step 7), and the tree re-encryption key generation for the link "new file/parent folder" (step 8). As in the general case, the encryption process outputs some meta-data MD_f and the encrypted file C_f, each one sent to the right entity (steps 10 to 13). The new tree re-encryption key is also added to the file meta-data (within step 12).

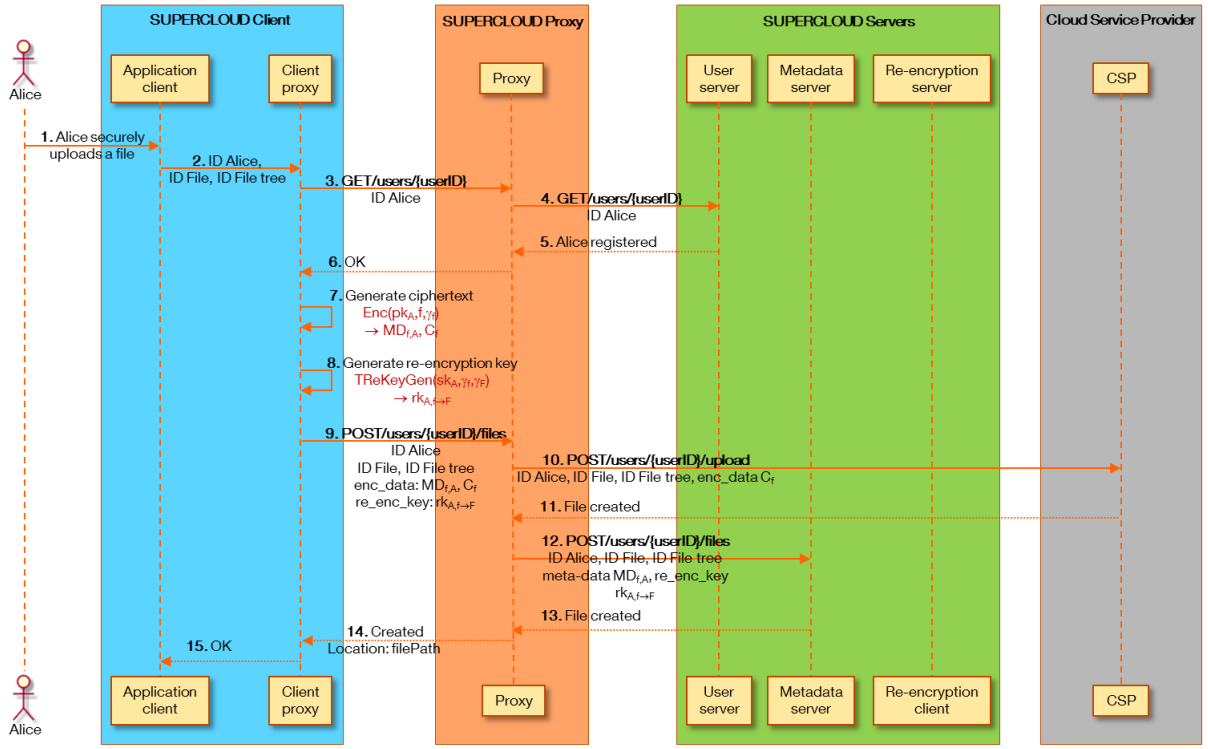


Figure 12.6: Upload using proxy re-encryption

File/folder sharing. We next consider the case where Alice wants to share a file or a folder with Bob. As shown in Figure 12.7, this step implies the execution of the user re-encryption key generation given above (step 15). The generated re-encryption key is then sent to the SUPERCLOUD WP3 dedicated server (step 19), and the new sharing is also sent to the Cloud Storage Provider (step 17) to let it know that the sharing exists, and that Bob can access the encrypted file.

File download. We focus on the case Bob wants to access a file owned by Alice, and for which a share exists. The resulting sequence diagram is given in Figure 12.8. As the file does not belong to Bob, the WP3 SUPERCLOUD proxy has first to re-encrypt the file (step 10), using the re-encryption process given previously. For this purpose, the proxy has to request the SUPERCLOUD servers to obtain the needed re-encryption keys (both tree and user, see step 9). This process only modifies the meta-data attached to the requested file. The encrypted file itself is simply requested to the corresponding Cloud Service Provider (steps 6 and 7). At the end of this re-encryption process, the proxy sends back to Bob the encrypted file and the modified meta-data (step 11). The latter can execute the decryption process given above (step 12), and obtain the file in clear.

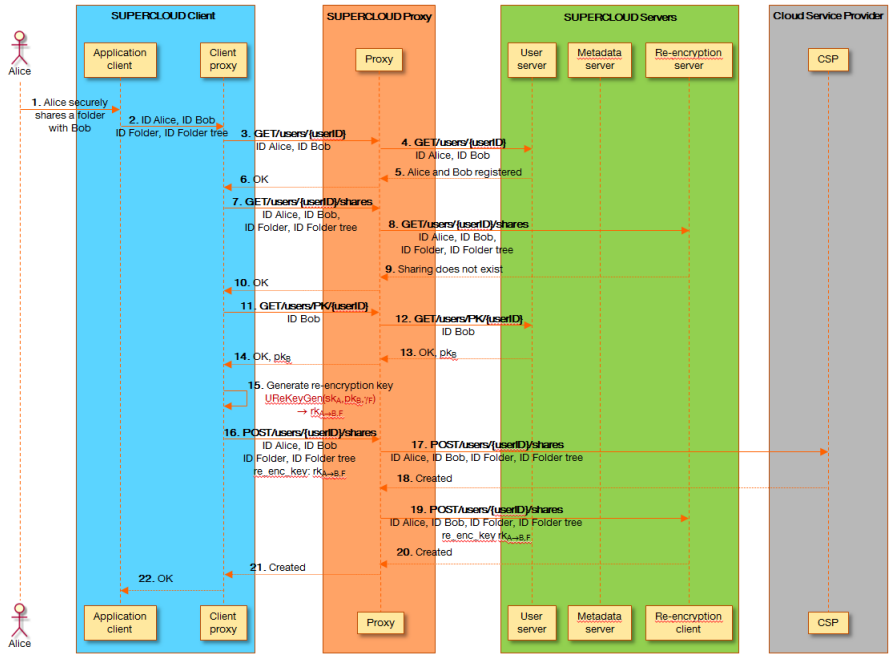


Figure 12.7: Sharing using proxy re-encryption

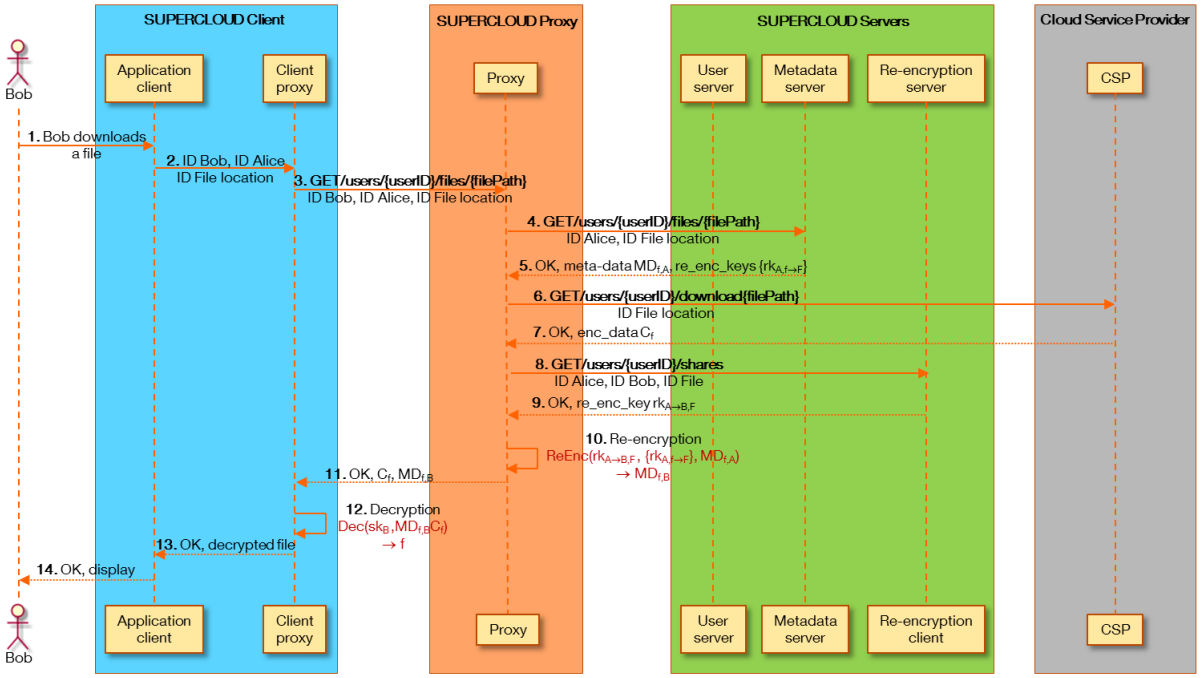


Figure 12.8: Download using proxy re-encryption

12.3.6 A First Implementation

We have implemented a first version of the above scheme, in a smartphone prototype. This is not the final implementation for SUPERCLOUD and the below figures will be updated during next year. This only gives a first overview of what will be possible.

Environment. The server side is a standard PC, with an Intel(R) Celeron(R) CPU E3300 at 2.50GHz.

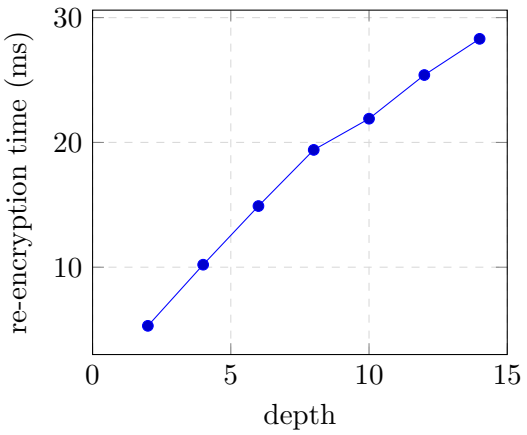


Figure 12.9: Re-encryption execution time per depth

The smartphone is a Samsung Galaxy S3. The implementations have been done in Java language, with some pre-computations for the cryptographic part.

Cryptographic algorithm. The cryptographic algorithm has been implemented using the elliptic curve variant of ElGamal, with a 128-bits security curve (namely the brainpool P256r1 one described in the IETF RFC 5639).
Efficiency. On the server/proxy side, the re-encryption process is related to the depth of the tree between the requested file and the shared folder (for example, in Figure 12.4, the depth between file $f_{2,1,1}$ and folder F_0 is equal to 3). With our experiment, we obtain a quasi-linear relation between the depth and the execution time, as shown in Figure 12.9.
From the smartphone's point of view, the decryption process is also close to a linear function in the tree depth. For a depth equal to 3, the decryption is done in less than 30 ms. With a depth of 15, we have nearly 120 ms. Other computations can be done in less than 10 ms and are independent in the depth of the file in the tree. The exact values are given in Table 12.1, for a depth equals to 3.

operation	time (in ms)
encryption	7.5
user re-encryption key generation	7.0
tree re-encryption key generation	8.0
decrypt with file rights	9.0
decrypt with upper folder rights	27.0

Table 12.1: Implementation results (for a depth = 3)

12.4 Attribute-based encryption

Another way to share data is to associate to each file an access control policy based on user's attributes. In case the data are encrypted, this is made possible by using a so-called attribute based encryption scheme, as explained in D3.1. We now focus on the way such cryptographic can be used in the context of SUPERCLOUD.

12.4.1 Attribute-Based Encryption in a Nutshell

Motivation. We consider a group wishes to put in place a system for its members, so that they can store and share sensitive documents, using a non-trusted Cloud Service Provider for storage. Each member of the group is first given some attributes, such as his/her role within the group (medical doctor, IT professional, ...), his/her living/working place, ... Based on that, an attribute-based encryption (ABE) scheme gives to anyone the possibility to encrypt and upload to the CSP a document, by choosing an access control policy related to attributes. Finally, if the attributes of a group member verify the access control policy embedded in the ciphertext, he/she will be able to decrypt the data. The Cloud Service Provider does not learn any information about the data in clear.
Description. Attribute-based encryption is an extension of traditional public key encryption in which the encryption and decryption phases are based on user's attributes. More precisely, we focus on *ciphertext-policy* ABE (CP-ABE) where the secret-key is associated to a set of attributes and the ciphertext is generated with an access policy. It then becomes feasible to decrypt a ciphertext only if one's attributes satisfy the used access policy. In this deliverable, we focus on Conjunctive Normal Form (CNF, i.e., with conjunctions (AND) of disjunctions (OR)) access policies.
An ABE scheme is composed of four algorithms, as detailed in D3.1. We here only recall the main characteristics and inputs/outputs, where κ is a security parameter and $\perp$ is the error message. As for PRE schemes, the parameters param are implicitly on input to all algorithms.

- $\mathsf{Setup}(\kappa) \rightarrow (\mathsf{param}, \mathsf{msk}, \mathsf{ek}, \mathcal{B})$.
- $\mathsf{KeyGen}(\mathsf{msk}, \mathcal{B}(u)) \rightarrow (\mathsf{sk}_u)$.
- $\mathsf{Enc}(\mathsf{ek}, m, \mathbb{A}) \rightarrow (C, \mathsf{Hdr})$.
- $\mathsf{Dec}(\mathsf{Hdr}), \mathsf{sk}_u, \mathcal{B}(u) \rightarrow m/\perp$.

We recall that $\mathcal{B}$ is the set of all possible client's attributes, u is a user having a set of attributes $\mathcal{B}(u) \subset \mathcal{B}$, and $\mathbb{A}$ is the access policy attached to the ciphertext.
Within the SUPERCLOUD WP3 architecture, the WP3 proxy is responsible for the KeyGen procedure and the WP3 servers manage the storage of file meta-data. Again, we now only sketched the cryptographic solution that is used. Details can be found in [91].

12.4.2 Main Ideas of the Scheme

The construction given in this document is based on two techniques. At first, we make use of the Junod-Karlov idea [187] to fight against attribute collusion (two users putting together their attributes to access a protected file each of them is not allowed to access alone). Secondly, we integrate the techniques from the multi-channel broadcast encryption (MCBE) scheme in [260] to obtain a ciphertext with a constant size. Note that the ideas in [187] and [260] are constructed from Boneh-Gentry-Waters (BGW) scheme [71].
More precisely, in [71], each element of the header has the form

$$\Big(g^r, (v \cdot \prod_{j \in \beta_k} g_{n+1-j})^r\Big),$$

where r is a random integer, and g, v and the g_j's are public group elements. Note that the proposed scheme inherits the use of a bilinear setting [153].
In the Junod-Karlov scheme [187], the authors manage to transform many instances of the BGW scheme [71] to an attribute-based encryption scheme, such that one instance of the BGW scheme corresponds to one clause in the CNF access policy. The resulting attribute-based encryption scheme then contains m BGW instances where m is the maximal number of clauses in the CNF access policy.

However, this leads to a ciphertext with $m+1$ parts. More precisely, for a CNF access policy $\mathbb{A} = \beta_1 \wedge \cdots \wedge \beta_m$, each component $\beta_k, k \in [m]$, is related to a BGW header as

$$\left(g^{rt_k}, (v^r \prod_{j \in \beta_k} g_{n+1-j}^r)^{t_k}\right).$$

In the MCBE scheme given in [260], the authors introduce a technique to multiply many BGW instances in one single value in order to support the new property of multi-channel for broadcast encryption. For this purpose, they introduce new integers x_j and provide a unique header given by

$$\left(g^r, \prod_{k=1}^{m} (v \cdot \prod_{j \in \beta_k} g_{n+1-j})^{r+\sum_{j \in \beta_k} x_j}\right).$$

Inspired by the technique given in [260], we manage to multiply the m instances of the BGW schemes to achieve an ABE scheme with constant-size ciphertext. The resulting ABE scheme therefore inherits the properties of the MCBE scheme, especially regarding compactness and security.

12.4.3 Sequence Diagrams

The sequence diagrams that are implemented for an ABE scheme are the ones given in Figures 12.1 and 12.2. We here only give some additional information.

Key generation. We notice that one can quite naturally adapt the techniques given in [59] and [103] to distribute the role of the central authority, managing msk, into several independent entities. This way, no external entity can know the secret keys sk_u of users.

File upload. The main point here is that the meta-data should include the header Hdr output by the ABE encryption process.

File download. Regarding the download phase, the client has to execute the ABE decryption phase and, if it has the correct attributes, its secret key sk_u would be able of decrypting the file and obtaining it in clear.

12.5 Conclusion

In this chapter, we have specified cryptographic tools that can be used to securely share data that are stored in an untrusted Cloud Service Provider. The aim of these tools is to let the user control his/her data by encrypting them before the upload. Then, the data can be shared, without needing to trust the CSP. In SUPERCLOUD, we propose two ways to proceed.

- *Proxy re-encryption scheme* permits to share the data with other users, by using their identities. Such solution is very mature and can easily be integrated into an existing platform. On client's side, there are two options. At first, a SUPERCLOUD library can be directly added to the end-user front-end. Such library is responsible for all cryptographic operations and requests to the SUPERCLOUD proxy. In this case, the front-end should implement the interface with the library. In the second option, the encryption/decryption components are also embedded in a specific Virtual Machine (SUPERCLOUD proxy and server), and the front-end has to request it to perform cryptographic operations. On that component, a first implementation has been done (see figures in Section 12.3.6) and the final version will be ready at the beginning of 2017.
- *Attribute-based encryption scheme* permits to share the stored data by using users' attributes. In this case, the solution that should be used depends on the context since it gives the kind (and number) of attributes that are needed, but also the kind of access control policy that

should be implemented (for example, the above specification is designed for CNF types access control policies). Besides, the integration of an ABE is quite similar to the one of a PRE and the two options are possible for the encryption/decryption processes. The implementation has just strated and we do not yet have figures. The components will be ready by mid 2017.

Chapter 13 Data Anonymization

In order to preserve individual's privacy while releasing personal data, anonymization techniques are commonly used to guarantee that the data subject can no longer be identified. Working on sensitive data requires therefore data anonymization. For that reason, datasets are processed in such a way that no inference in respect to user's identity can be made or rather assign revealed data to a natural person. There are several techniques known that can be applied to achieve data anonymization, such as *Perturbation*, *Encryption* of personally identifiable information or *k-anonymity* including among others *Generalization*. Within SUPERCLOUD we mainly focused on *k-anonymity* concerning data anonymization as already mentioned in the previously released SUPERCLOUD deliverable D3.1 [308]. Therefore, data anonymization, especially k-anonymity, was determined as a part of an alternative architecture approach. For this architecture proposal, k-anonymity was selected as an additional security component in combination with Multi-Party Computation (MPC). The combination of MPC with k-anonymity prevents from storing data in plaintext inside one of the SUPERCLOUD servers. If any data are needed, such as for statistical purposes (for a read-only recipient), the data anonymization (k-anonymity) will be performed. More precisely, when the data shares (produced by MPC) are re-computed again, the anonymization of the records will take place directly after to avoid that an adversary sees a plain record. This anonymization is performed on the side of SUPERCLOUD's architecture, as shown in Figure 13.1.

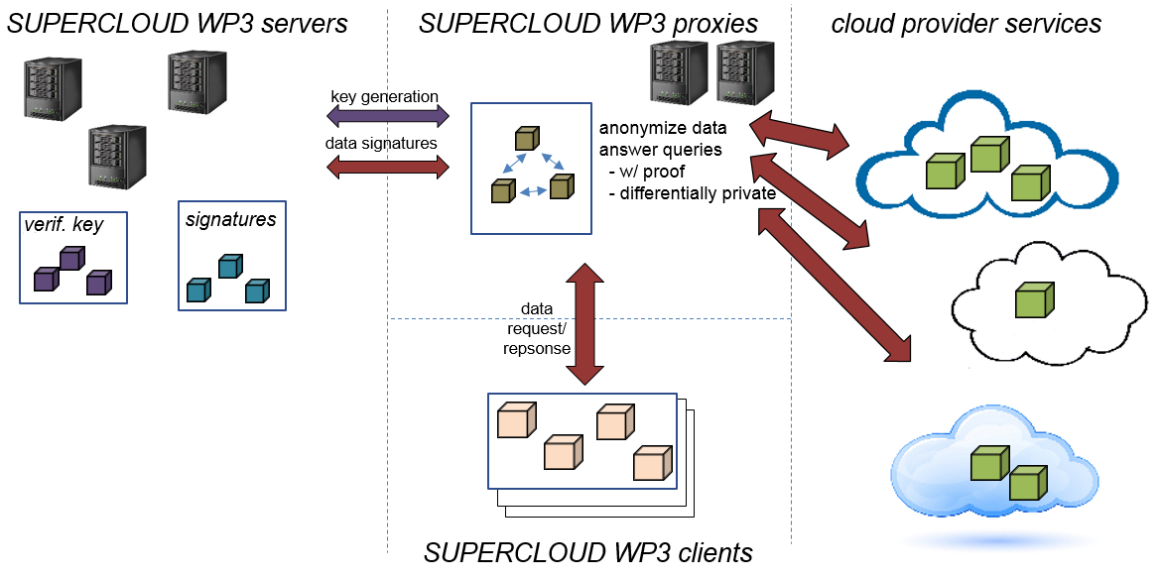


Figure 13.1: Modified WP3 architecture based on MPC and k-anonymity [308]

Within this chapter, we will go into detail on k-anonymity and related algorithms in order to set-up a basis for the upcoming implementation work within the next deliverable D3.3 of SUPERCLOUD.

13.1 K-anonymity

The focus of anonymization lies on the irreversibility of the released sensitive data. In order to release data without any disclosure of sensitive information, several techniques, such as *Generalization* (remove specificity by replacing an attribute by a more general value), *Perturbation* (change data of record in a statistically insignificant way), *Encryption* of personally identifiable information or *k-anonymity*, are commonly used. As already mentioned, within this chapter we will mainly focus on k-anonymity, as this is among the most relevant anonymization technique for the SUPERCLOUD use cases dedicated to health services and their data anonymization.
In general, data may contain different kind of attributes/identifiers: *explicit identifiers* (clearly identify individuals, e.g. by name), *sensitive attributes* (e.g. salary, disease) and *quasi-identifiers* (attributes in combination could lead to data disclosure / re-identification).
The main goal of k-anonymity is to enable the opening of data, where each record is indistinguishable from at least $k-1$ other records with respect to the quasi-identifier (QI). This implies that the anonymity of the released data is satisfied, if each sequence of values k in the respective data table occurs at least k-times. In this case, the k can be replaced by an integer. Therefore, if a 4-anonymity was reached, it implies that the attributes representing the quasi-identifier can be found in at least four rows.
K-anonymity itself is comprised by the already introduced anonymization technique *Generalization* and the so-called *Suppression* method. While the former is responsible for the generalization of the quasi-identifier attributes with less specific values until k-identical values are reached, the latter is deleting uniquely identifying attributes (explicit identifiers), like names among others, in order to avoid information disclosure.

Table 13.1 demonstrates briefly the principle of k-anonymity based on generalization and suppression performed on sample health data. In the ensuing table, the quasi-identifier is composed of *age*, *gender* and *ZIP code*. While table (a) represents the plain health data including explicit identifiers (first and last name of patient), which would reveal sensitive information of patients in case of data opening, table (b) depicts the anonymized health data based on the chosen quasi-identifier. In detail, suppression (eliminating of the explicit identifier *Name*) and a generalization (of *age* and *ZIP code*) was performed. As seen in table (b), *age* as well as *ZIP code* was generalized by one level in order to avoid the irreversibility and provide privacy-preserved data with minimal information loss. Within this anonymization example, the maximal anonymity amounts $k = 2$, as it is implied by the definition of k-anonymity, respectively the occurrence of the sequence of values.

Table 13.1: Usage of k-anonymity on untreated plain health data (a) by means of a quasi-identifier QI = [*age, gender, ZIP code*], which results in a 2-anonymity, seen in table (b)

Name	Age	Gender	ZIP Code	Objection
Damien Luongo	26	Male	22145	Short Breath
Jamar Rollinson	28	Male	22147	Chest Pain
Indira Lindon	28	Female	22131	Hypertension
Lizette Atchison	25	Female	22133	Hypertension
Rhonda Paisley	41	Female	22133	Obesity
Carie Casselman	44	Female	22135	Chest Pain
Deeann Goldschmidt	63	Male	22131	Chest Pain
Charles Hardegree	62	Male	22131	Obesity
Eddy Luedtke	60	Male	22132	Short Breath
Casandra Padro	15	Female	22144	Chest Pain
Kerstin Fullerton	18	Female	22143	Chest Pain

(a) Plain health data

Tuple	Age	Gender	ZIP Code	Objection	k
t1	25-49	Male	2214*	Short Breath	2
	25-49	Male	2214*	Chest Pain	
t2	25-49	Female	2213*	Hypertension	4
	25-49	Female	2213*	Hypertension	
	25-49	Female	2213*	Obesity	
	25-49	Female	2213*	Chest Pain	
t3	50-74	Male	2213*	Chest Pain	3
	50-74	Male	2213*	Obesity	
	50-74	Male	2213*	Short Breath	
t4	0-24	Female	2214*	Chest Pain	2
	0-24	Female	2214*	Chest Pain	

(b) Anonymized health data

The figures presented below (Figure 13.2 and 13.3) represent a step-by-step generalization of the attributes *age*, *gender* and *ZIP code* based on the previous depicted data in Table 13.1. Figure 13.2 depicts the generalization of the attribute *age*, whereby the values are incrementally generalized three times until the highest level of the generalization (total generalization) is reached (age 0-99). Whereas the attribute *gender* can only be generalized once. However, the attribute *ZIP code*, a five-digit zip-code, is generalized five times until the total generalization is reached. As a result of this, there are two main noticeable generalization types: on the one hand, an incremental generalization of the age, whereby the value is generalized into a more general range of values (ages), as it can be seen in Figure 13.2. On the other hand, a sequential generalization, which represents a replacement of the last non-generalized character of the attribute, as seen in Figure 13.3.

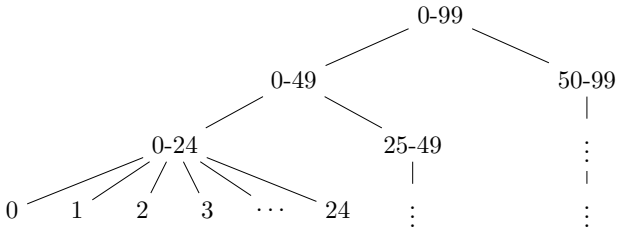
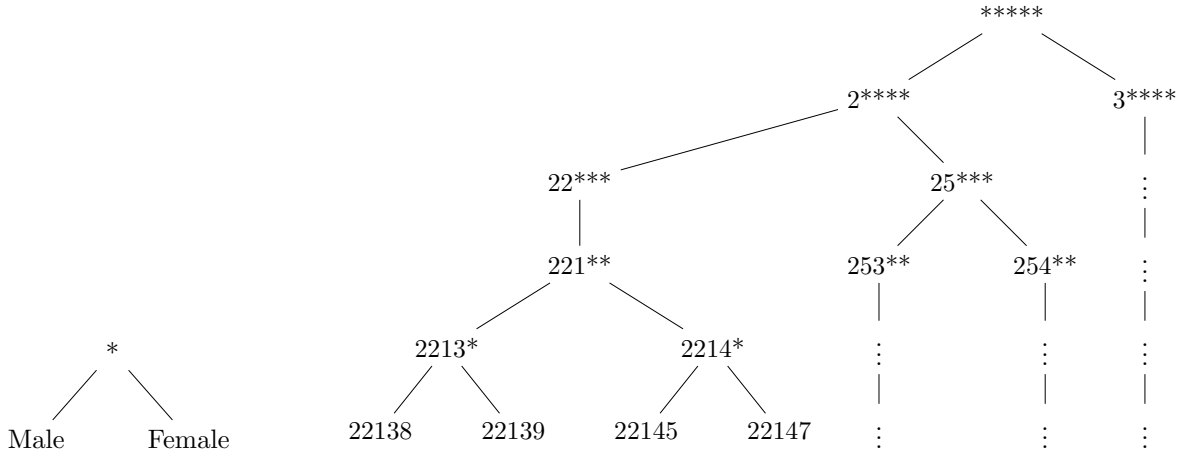


Figure 13.2: 3-stage incremental generalization of the attribute *age*



(a) Gender generalization

(b) ZIP code generalization

Figure 13.3: Sequential generalization of the attributes *gender* and *ZIP-code* respectively

13.1.1 Detailed Procedure

The principle and procedure of anonymization, respectively the generalization of a data table, is straightforward. First of all, the quasi-identifier, based on various attributes, has to be selected. Upon this, the generalization of the data table can be done (on the basis of the quasi-identifier) and a so-called lattice can be built. The lattice represents stepped generalization of the data in the form of a node list. A graphical representation of a lattice based on the health data depicted in Table 13.1 and the quasi-identifier QI = [*Age, Gender, ZIP Code*] can be seen below in Figure 13.4. The lattice itself is based on the generalization of each attribute, which composes the quasi-identifier. The top node of the lattice represents the total anonymization, whereas the data, respectively the value behind, does not contain useful information anymore. The lattice is an important tool within the generalization to find the globally optimal solution with minimal information loss. There are so-called cost metrics

available in order to select the optimal solution, described in detail in section 13.2. As the traverse of the lattice represents the costly part of the anonymization procedure, there exist several algorithms, which traverse and prune efficiently the lattice in order to find the globally optimal solution. The most efficient algorithm is the *Optimal Lattice Anonymization* [143] algorithm and is described in detail in section 13.3.

Figure 13.4 represents the lattice based on the plain health data depicted in Table 13.1 composed on the basis of the quasi-identifier QI = [*Age, Gender, ZIP Code*]. In order to compose completely a lattice, it is important to generalize each quasi-identifier attributes until its total anonymization is achieved.

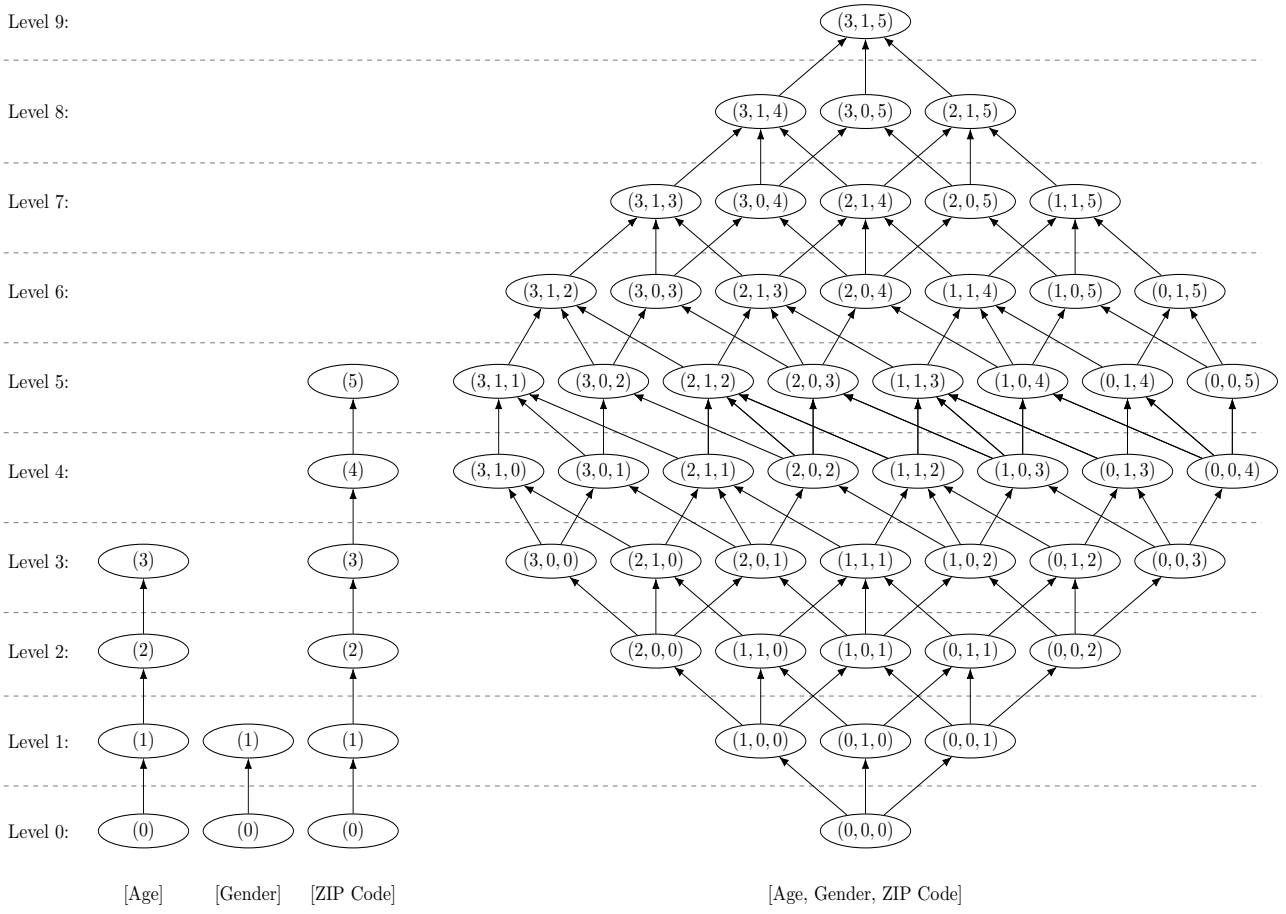


Figure 13.4: Composition of a lattice based on the quasi-identifier QI = [*Age, Gender, ZIP Code*]

13.2 Cost Metric

The more the data will be generalized, the higher will be the information loss. Hence to keep the loss of information as low as possible, so-called cost metrics are used for the measurement of information loss. There are several metrics known, such as the distance measurement proposed by Samarati [275] and precision cost metric published in [293] by Sweeney. According Samarati's distance cost metric, the optimal solution is given by the node with the lowest lattice height. However, this solution does not constitute a good cost metric for information loss measurement, because it does not account for

the generalization hierarchy depths of the QIs. On the contrary, the precision cost metric by Sweeney considers the height and depth of the generalization hierarchy. Therefore, the ratio of the number of applied generalization steps to the total number of (possible) generalization steps will be calculated in order to provide the precision for a specific generalization. The ratio for the precision cost metric for $QI = [\, QI_1, ..., QI_n \,]$ is calculated as follows:

$$Prec\,(\, QI_1, ..., QI_n \,) = (\, Level\,[\, QI_1 \,] \,/\, Max\,[\, QI_1 \,] + ... + Level\,[\, QI_n \,] \,/\, Max\,[\, QI_n \,] \,) \,/\, n$$

As a result, the precision of the first bottom node or rather the start node is zero, respectively the information loss is 0%, whereas the precision of the top node is one, respectively the information loss is 100%. This implies that the higher the precision of a node is, the greater the information loss (or equivalently the lower the precision the lesser the information loss).

13.3 Optimal Lattice Anonymization Algorithm

Besides the main goal of k-anonymity concerning the de-identification of data, the focus lies on the search of the optimally generalized node that satisfies the k-anonymity with minimal information loss (or with maximal information content respectively). Since the seminal paper of Sweeney [293] concerning the invention of k-anonymity, several improvements were proposed, such as an algorithm provided by Samarati [275] and the optimized approach called Optimal Lattice Algorithm (OLA) [143] of few years later. The OLA approach was proposed by El Emam et al. in 2009 and is based on the divide-and-conquer technique in order to divide a lattice into smaller sub lattices. By means of binary search, the OLA algorithm is used to find optimal nodes (k-minimal nodes) for each sub-lattices, while keeping the efficiency high and information loss low. The particular optimization in contrast to previously proposed algorithms is the usage of predictive tagging to prune parts of the search space. The OLA algorithm proceeds in three steps:

1. Choose a path (sub-lattice), whereby all generalizations of the path are untagged
2. Apply binary search on the sub-lattice in order to find the local optimal k-anonymous node and store all located k-minimal nodes.
3. Compare all located k-minimal nodes concerning their precision cost metric in order to select the globally optimal solution with the smallest information loss

The pseudo code *Algorithm* 13 covers the main procedure of the above-described Optimal Lattice Anonymization algorithm in an abstract way. Based on the three steps of the OLA algorithm, *Algorithm* 13 is composed of three separated functions.
The first function *getGloballyOptimalSolution* requires the lattice built previously as input, which represents in general a bottom-up generalization of the data. Further on, the chosen k, respectively the minimum anonymization, which has to be reached in order to open the data without de-identification, is necessary.
The function is based on two loops. While the first loop iterates over all levels of the provided lattice in order to call the function *findPath* as well as the function *findKMinimalNodes*, the second loop iterates over all found k-minimal nodes to get the globally optimal solution, which constitutes the returning value of this function. As a result, the first loop of this algorithm embraces the first two steps of the OLA algorithm procedure as described in the section above.
The final outcome of this function is the globally optimal solution for the provided lattice and is represented in the form of a specific node, which satisfies the k-anonymity while minimal information loss.

The function *findPath* is responsible for finding a path within the lattice, where all generalizations of this path are not classified or untagged. If the transferred node (as parameter) is not already the end of the lattice, then this node will be added into the path-nodelist. Further on, by means of the function *getNextNode*, the next untagged node for a given node within the lattice will be returned. As the lattice has to be traversed completely in order to select a path successfully, the algorithm calls itself in a recursive way to finally return the path.
The last function *findKMinimalNodes* within the OLA *Algorithm* 13 is used in order to find all k-minimal nodes, which in turn are needed to determine the globally optimal solution by means of precision metric. As mentioned in the previous paragraph, the Optimal Lattice Anonymization algorithm is using the binary search that constitutes the core of the *findKMinimalNodes* function below. Furthermore, the focus of this function lies in the decision-making process concerning the level of the anonymization. To be more accurate, the function is responsible for the validation, if the transferred k equals the anonymization of the entrance node or not. Based on the following two decisions, different action points are triggered:

- If the node on the entrance level (calculated by means of binary search) is equal or greater than chosen and transferred k, then:
  - Store the actual node on the entrance level as a possible k-minimal solution for this path
  - Perform predictive tagging on all generalizations above the entrance node up to the top (end) of the path
  - Recursive function-call with the lower part of the path up to the node on the entrance level in order to check, if there is a better k-minimal solution with less information loss there
- If the node on the entrance level (calculated by means of binary search) is less than chosen and transferred k, then:
  - Perform predictive tagging on all pre-generalizations below the entrance node down to the bottom (beginning) of the path
  - Recursive function-call with the upper part of the path up to the end node of the path in order to prune further parts of the path

Figure 13.5 below sums up the subject matter of the k-anonymity principle including the cost metric calculation and the described OLA algorithm. It illustrates the calculated precisions as well as the degree of anonymity for each node up to the 3^{rd} level of the depicted lattice (Figure 13.4) based on the provided health data (Table 13.1). If we take only the first three levels into consideration and proceed with the assumption that we only accept anonymity 2 and higher, the following four nodes will be selected by the OLA algorithm as the possible k-minimal nodes, respectively as the candidates for the optimal solution of the given lattice: $(1, 0, 1)$; $(2, 0, 1)$; $(1, 1, 1)$; $(1, 0, 2)$. In further consequence, the calculated precision of the k-minimal nodes is crucial for the further procedure of the algorithm. Therefore, the node with the lowest precision, respectively with the lowest information loss, will be selected by the algorithm as the globally optimal solution. In case of the given lattice, the globally optimal solution is represented by the node $(1, 0, 1)$, as seen highlighted in Figure 13.5. On closer examination, Table 13.1 depicts the result of a 2-anonymity data anonymization, whereby the QI is represented by the resulting globally optimal solution node $(1, 0, 1)$, which results in turn in a one-step incremental generalization of the attribute *age* as well as a one-step sequential generalization of the attribute *ZIP code*.

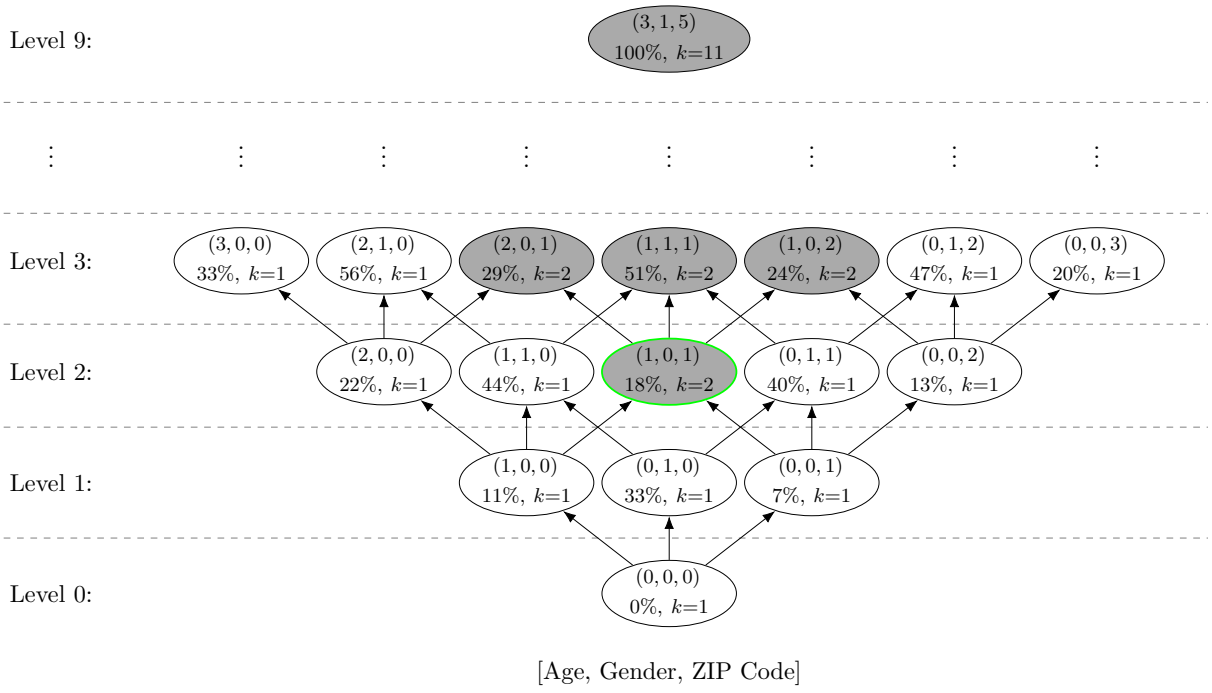


Figure 13.5: Graphical representation of a globally optimal solution assuming a 2-anonymity based on the quasi-identifier QI = [*Age, Gender, ZIP Code*] considering only first three levels of lattice

Algorithm 13: Optimal Lattice Anonymization (OLA) Algorithm

```
1  Function getGloballyOptimalSolution ( Lattice lattice, Number k ) begin
       input  : lattice, k^th anonymity
       output: globally optimal solution
2      path ← new NodeList
3      kMinimalNodes ← new List
4      potentialSolution ← new Node
5      highestPrecision ← new Double
6      globalOptimalSolution ← new Node
7      startNode ← lattice.firstElement
8      foreach level in lattice.level do
9          path ← findPath ( lattice, startNode, path )
10         kMinimalNodes.add ( findKMinimalNodes ( path, k, path.firstElement, path.lastElement,
             potentialSolution ) )
11         startNode ← getNextNode ( startNode )
12     foreach node in kMinimalNodes do
13         precision ← calcPrecision ( node )
14         if precision > highestPrecision then
15             highestPrecision ← precision
16             globalOptimalSolution ← node
17     return globalOptimalSolution
18 Function findPath ( Lattice lattice, Node startNode, NodeList path ) begin
       input  : lattice, start node, path (sub-lattice)
       output: untagged path (sub-lattice)
19     if startNode ≠ lattice.endNode AND !startNode.tagged then
20         path.add ( startNode )
21         nextNode ← getNextNode ( startNode )
22         return findPath ( lattice, nextNode, path )
23     else
24         return path
25 Function findKMinimalNodes ( NodeList path, Number k, Node startNode, Node endNode,
     Node potentialSolution ) begin
       input  : path (sub-lattice), k, start node, end node, potential solution (initially null)
       output: k-minimal solution for given path
26     entranceLevel ← ⌊ ( startNode.level + endNode.level) / 2 ⌋
27     if path.node [ entranceLevel ].anonymity ≥ k then
28         potentialSolution = path.node [ entranceLevel ]
29         predictiveTagging ( path, path.node [ entranceLevel + 1 ], path.lastElement )
30         return findKMinimalNodes ( path, k, startNode, path.node [ entranceLevel ],
             potentialSolution )
31     else if path.node [ entranceLevel ].anonymity < k then
32         predictiveTagging ( path, startNode, path.node [ entranceLevel ] )
33         return findKMinimalNodes ( path, k, path.node [ entranceLevel + 1 ], path.lastElement,
             potentialSolution )
34     return potentialSolution
```

13.4 Conclusion

In this chapter we have presented an in-depth look into the subject matter of data anonymization techniques, especially of those of k-anonymity. The main goal of data anonymization is defined by the irreversibility while releasing sensitive data. Therefore, anonymization techniques, such as k-anonymity, have to be applied on the plain data in order to avoid any information disclosure and preserve the individual's privacy while data opening. Generalization of the quasi-identifier attributes and suppression of explicit identifiers constitutes the basis of k-anonymity. K-anonymity was already introduced briefly and established as a part of an alternative architecture approach in D3.1 [308]. Through the application of k-anonymity, anonymized data can be provided to read-only users without revealing any sensitive information.
There are several algorithms known in order to traverse a lattice. The lattice itself depicts the result of a step-by-step generalization of the attributes representing the quasi-identifier. Among others, we have described the Optical Lattice Anonymization algorithm invented by El Emam et al. [143]. The OLA algorithm represents the most efficient algorithm concerning lattice traversal, which is generally based on the divide-and-conquer technique. The decision-making process is in turn based on the previously described precision cost metric calculation proposed by Sweeney [293].
The described OLA algorithm 13 and the related and depicted pseudo code in section 13.3 should state a clear basis for the upcoming implementation work within the next deliverable. The consequent prototype will be interconnected with the determined SUPERCLOUD architecture and its cloud storage principle.

Chapter 14 Conclusion and Future Work

In this deliverable we have presented the security and dependability components for SUPERCLOUD data management solution. The deliverable first defined novel components pertaining to state-machine replication, which will be used to replicate critical pieces of SUPERCLOUD metadata across multiple clouds, orchestrated with Hyperledger fabric. Then, we covered SUPERCLOUD distributed storage solutions which will be used use to manage bulk data. These solutions will be orchestrated around Janus user-centric multi-cloud storage. Finally, the third part describes advanced data security components, focusing, in particular on data privacy techniques. These components are envisioned to be used orthogonally within Janus and Hyperledger fabric. The deliverable also provides a high level overview on planned integration.

Whenever the state of maturity of our prototypes allowed so, we included detailed benchmarks and evaluation of proposed components. Several components described in this deliverable have already been published in top research conferences.

As future work, we will be working towards integrating a subset of these components into a proof-of concept prototype (D3.3, M28) and the final prototype (D3.4, M36). Along the way we will be refining and updating our designs and solutions and considering their mutual integration as well as integration with other work packages.

Bibliography

[1] TPC-C Benchmark: http://tpc.org.

[2] Amazon S3. http://aws.amazon.com/s3/.

[3] Amazon S3 pricing. https://aws.amazon.com/s3/pricing/.

[4] BenchmarkSQL. https://bitbucket.org/openscg/benchmarksql.

[5] bft-smart. http://code.google.com/p/bft-smart/.

[6] Cassandra documentation. http://www.datastax.com/documentation/cassandra/2.0/cassandra/gettingStartedCassandraIntro.html.

[7] CockroachDB Design Document. https://github.com/cockroachdb/cockroach/blob/master/docs/design.md.

[8] DepSky webpage. http://cloud-of-clouds.github.io/depsky/.

[9] FUSE-J. http://fuse-j.sourceforge.net/.

[10] Google storage. https://developers.google.com/storage/.

[11] HydraBase – The evolution of HBaseFacebook. https://code.facebook.com/posts/321111638043166/hydrabase-the-evolution-of-hbase-facebook/.

[12] Java TPC-C. https://github.com/AgilData/tpcc.

[13] Kryo - Java serialization and cloning: fast, efficient, automatic. https://github.com/EsotericSoftware/kryo.

[14] Microsoft Azure Site Recovery. https://azure.microsoft.com/en-us/services/site-recovery/.

[15] MWMR-registers webpage. https://github.com/cloud-of-clouds/mwmr-registers/.

[16] MySQL - The InnoDB Storage Engine. http://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html.

[17] MySQL 5.7 documentation. http://dev.mysql.com/doc/refman/5.7/en/.

[18] MySQL replication. http://dev.mysql.com/doc/refman/5.7/en/replication.html.

[19] PostgreSQL. http://www.postgresql.org/.

[20] PostgreSQL Documentation. http://www.postgresql.org/docs/.

[21] Rackspace cloud files. http://www.rackspace.co.uk/cloud/files.

[22] Softlayer Cloud Storage. http://www.softlayer.com/Cloud-storage/.

[23] The rsync algorithm. http://rsync.samba.org/tech_report/tech_report.html.

[24] VMware vCloud Air Disaster Recovery. https://www.vmware.com/cloud-services/infrastructure/vcloud-air-disaster-recovery.

[25] Zmanda recovery manager for MySQL. http://www.zmanda.com/.

[26] Evaluating color descriptors for object and scene recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32, 2010.

[27] Notesecure image retrieval thros on non-interactive secure comparison in image feature extraction in the encrypted domain with privacy-preserving sift. 2014.

[28] Business continuity trends and challenges 2016. http://www.continuitycentral.com/index.php/news/business-continuity-news/776-business-continuity-trends-and-challenges-2016, January 2016.

[29] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5), 2006.

[30] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. RACS: A case for cloud storage diversity. *SoCC*, 2010.

[31] Atul Adya *et al.* Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, 2002.

[32] M. Aguilera, B. Englert, and E. Gafni. On using network attached disks as shared memory. In *Proc. of the PODC*, 2003.

[33] James Alderman, Christian Janson, Carlos Cid, and Jason Crampton. Access Control in Publicly Verifiable Outsourced Computation. In *Proc. ASIACCS*, 2015.

[34] Bowen Alpern and FredB. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[35] 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.

[36] Y. Amir and A. Wool. Evaluating quorum systems over the internet. In *Proc. of the 26th Int. Symposium on Fault-Tolerant Computing*, 1996.

[37] Yair Amir, Brian A. Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577, 2011.

[38] Prabhanjan Ananth, Nishanth Chandran, Vipul Goyal, Bhavana Kanukurthi, and Rafail Ostrovsky. Achieving Privacy in Verifiable Computation with Multiple Servers - Without FHE and without Pre-processing. In *Proceedings of PKC*, 2014.

[39] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Trans. on Computer Systems*, 14(1):41–79, February 1996.

[40] Tal Anker, Danny Dolev, Gregory Greenman, and Ilya Shnayderman. Evaluating total order algorithms in WAN. In *In Proc. of the Int. Workshop on Large-Scale Group Communication*, 2003.

[41] Anonymous. Details omitted for double-blind reviewing. Reviewers can obtain a copy through USENIX OSDI 2016 Program Chairs. Technical report.

[42] ARM. ARM security technology – Building a secure system using TrustZone technology. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`, 2009.

[43] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1), 1995.

[44] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, January 2015.

[45] Shai Avidan and Moshe Butman. Blind vision. In *9th European Conference on Computer Vision*, ECCV, 2006.

[46] Y. Bai, L. Zhuo, B. Cheng, and Y. F. Peng. Surf feature extraction in encrypted domain. In *IEEE International Conference on Multimedia and Expo*, ICME, 2014.

[47] Peter Bailis and Kyle Kingsbury. The network is reliable: An informal survey of real-world communications failures. *ACM Queue*, 2014.

[48] Sumeet Bajaj and Radu Sion. Trusteddb: A trusted hardware-based database with privacy and data confidentiality. In *IEEE Transactions on Information Forensics and Security*, 2014.

[49] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[50] Omar Bakr and Idit Keidar. Evaluating the running time of a communication round over the internet. In *Proceedings of the 21st Symposium on Principles of Distributed Computing*, 2002.

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

[52] M. Barni, P. Failla, R. Lazzeretti, A. R. Sadeghi, and T. Schneider. Privacy-preserving ecg classification with branching programs and neural networks. *IEEE Transactions on Information Forensics and Security*, 6, 2011.

[53] Mauro Barni, Tiziano Bianchi, Dario Catalano, Mario Di Raimondo, Ruggero Donida Labati, Pierluigi Failla, Dario Fiore, Riccardo Lazzeretti, Vincenzo Piuri, Fabio Scotti, and Alessandro Piva. Privacy-preserving fingercode authentication. In *Proceedings of the 12th ACM Workshop on Multimedia and Security*, MM Sec, 2010.

[54] C. Basescu et al. Robust data sharing with key-value stores. In *Proc. of the DSN*, 2012.

[55] Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly Auditable Secure Multi-Party Computation. In *Proceedings of SCN*, 2014.

[56] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *USENIX Symposium on Operating Systems*, 2014.

[57] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *9th European Conference on Computer Vision*, ECCV, 2006.

[58] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Consensus-oriented parallelization: How to earn your first million. In *Proceedings of the 16th ACM/IFIP/USENIX Middleware Conference – Middleware'15*, 2015.

[59] Mira Belenkiy, Jan Camenisch, Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Hovav Shacham. Randomizable proofs and delegatable anonymous credentials. In *CRYPTO 2009*, volume 5677 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2009.

[60] R. Bellafqira, G. Coatrieux, D. Bouslimi, and G. Quellec. Content-based image retrieval in homomorphic encryption domain. In *IEEE Engineering in Medicine and Biology Society*, EMBC, 2015.

[61] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Message-locked encryption and secure deduplication. In *EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 296–312. Springer, 2013.

[62] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness Theorems for Non-cryptographic Fault-tolerant Distributed Computation. In *Proceedings of STOC*, 1988.

[63] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *Proceedings of CRYPTO*. 2013.

[64] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Towards optimal distributed consensus. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 410–415, 1989.

[65] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage*, 9(4), 2013.

[66] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. SCFS: a shared cloud-backed file system. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC '14)*, 2014.

[67] Alysson Bessani, Marcel Santos, Joo 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.

[68] Alysson Bessani, Joao Sousa, and Eduardo Alchieri. State machine replication for the masses with BFT-SMART. In *Proc. of the 44th IEEE/IFIP Int. Conference on Dependable Systems and Networks*, 2014.

[69] M. Blaze, G. Bleumer, and M. Strauss. Divertible protocols and atomic proxy cryptography. In *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 127–144. Springer, 1998.

[70] 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 *Proc. of the USENIX Symposium on Networked Systems Design and Implementation*, April 2011.

[71] Dan Boneh, Craig Gentry, and Brent Waters. Collusion resistant broadcast encryption with short ciphertexts and private keys. In *CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 258–275. Springer, 2005.

[72] Anna Bosch, Andrew Zisserman, and Xavier Muñoz. Scene classification via plsa. In *9th European Conference on Computer Vision*, ECCV, 2006.

[73] Anna Bosch, Andrew Zisserman, and Xavier Muñoz. Image classification using random forests and ferns. In *IEEE International Conference on Computer Vision*, ICCV, 2007.

[74] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[75] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008.

[76] Peter Brouwer. The art of data replication. Oracle Technical White Paper, 2011.

[77] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. In *Distributed Systems (2nd Ed.)*. ACM Press & Addison-Wesley, New York, 1993.

[78] Gertjan J. Burghouts and Jan-Mark Geusebroek. Performance evaluation of local colour invariants. *Computer Vision and Image Understanding*, 113, 2009.

[79] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[80] Christian Cachin. Distributing trust on the Internet. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 183–192, 2001.

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

[82] Christian Cachin, Birgit Junker, and Alessandro Sorniotti. On limitations of using cloud storage for data replication. In *Proc. 6th Workshop on Recent Advances in Intrusion Tolerance and reSilience – WRAITS'12*, 2012.

[83] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols (extended abstract). In *Advances in Cryptology: CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001.

[84] Christian Cachin and Jonathan A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proc. International Conference on Dependable Systems and Networks (DSN-DCCS)*, pages 167–176, June 2002.

[85] Christian Cachin, Simon Schubert, and Marko Vukolić. Non-determinism in Byzantine fault-tolerant replication. e-print, arXiv:1603.07351 [cs.DC], 2016.

[86] Christian Cachin and Stefano Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. In *Proc. of the DSN*, 2006.

[87] Brad Calder, Ju Wang, Aaron Ogus, et al. Windows Azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[88] Brad Calder et. al. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles – SOSP'11*, 2011.

[89] S. Canard and J. Devigne. Highly privacy-protecting data sharing in a tree structure. *Journal of Future Generation Computer Systems*, 62:119–127, 2016.

[90] S. Canard, J. Devigne, and F. Laguillaumie. Improving the security of an efficient unidirectional proxy re-encryption scheme. *Journal of Internet Services and Information Security (JISIS)*, 1(2/3):140–160, 8 2011.

[91] Sébastien Canard and Viet Cuong Trinh. Private ciphertext-policy attribute-based encryption schemes with constant-size ciphertext supporting cnf access policy. Cryptology ePrint Archive, Report 2015/891, 2015. `http://eprint.iacr.org/2015/891`.

[92] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. *IACR Cryptology ePrint Archive*, 2000:67, 2000.

[93] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. In *Proceedings of CRYPTO 2015*, pages 3–22, 2015.

[94] Mark Carlson et. al. Software defined storage. Technical report, SNIA - Storage Networking Industry Association, 2015.

[95] Henry Carter, Charles Lever, and Patrick Traynor. Whitewash: outsourcing garbled circuit generation for mobile devices. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, pages 266–275, 2014.

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

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

[98] Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. Middleware-based database replication: The gaps between theory and practice. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008.

[99] Rafal Cegiela. Selecting technology for disaster recovery. In *International Conference on Dependability of Computer Systems (DepCos-RELCOMEX'06)*, 2006.

[100] Tushar Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live - An engineering perspective. In *Proc. of the 26th ACM Symposium on Principles of Distributed Computing*, 2007.

[101] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

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

[103] Melissa Chase and Sherman S. M. Chow. Improving privacy and security in multi-authority attribute-based encryption. In *ACM Conference on Computer and Communications Security, CCS 2009*, pages 121–130. ACM, 2009.

[104] B. Chevallier-Mames, P. Paillier, and D. Pointcheval. Encoding-free El Gamal encryption without random oracles. In *PKC'06*, volume 3958 of *LNCS*, pages 91–104. Springer, 2006.

[105] Alessandro Chiesa, Eran Tromer, and Madars Virza. Cluster Computing in Zero Knowledge. In *Proceedings of EUROCRYPT*, 2015.

[106] G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. *Distributed Computing*, 18(1), 2005.

[107] Gregory V. Chockler, Dan Dobre, Alexander Shraer, and Alexander Spiegelman. Space bounds for reliable multi-writer data store: Inherent cost of read/write primitives. In *Proc. of the PODC*, 2016.

[108] Seung Geol Choi, Jonathan Katz, Ranjit Kumaresan, and Carlos Cid. Multi-client non-interactive verifiable computation. In *TCC*, pages 499–518, 2013.

[109] Jia-Kai Chou, Chuan-Kai Yang, and Hsing-Ching Chang. Encryption domain content-based image retrieval and convolution through a block-based transformation algorithm. *Multimedia Tools and Applications*, 74, 2015.

[110] S. S. M. Chow, J. Weng, Y. Yang, and R. H. Deng. Efficient unidirectional proxy re-encryption. In *AFRICACRYPT'10*, volume 6055 of *LNCS*, pages 316–332. Springer, 2010.

[111] Jae Yoon Chung, Carlee Joe-Wong, Sangtae Ha, James Won-Ki Hong, and Mung Chiang. CYRUS: Towards client-defined cloud storage. In *Proc. of the 10th ACM European Systems Conference – EuroSys'15*, 2015.

[112] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 277–290, New York, NY, USA, 2009. ACM.

[113] Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proc. 6th Symp. Networked Systems Design and Implementation (NSDI)*, pages 153–168, 2009.

[114] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proceedings of ACM/SIGOPS Symposium on Operating Systems Principles (SOSP'13)*, 2013.

[115] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[116] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[117] James C. Corbett et. al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems*, 31(3):8:1–8:22, 2013.

[118] Miguel Correia, Daniel Gómez Ferro, Flavio P. Junqueira, and Marco Serafini. Practical hardening of crash-tolerant systems. In *USENIX ATC'12*, 2012.

[119] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, SRDS '04, pages 174–183. IEEE Computer Society, 2004.

[120] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Information and Computation*, 118(1):158–179, 1995.

[121] Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, João Paulo, José Pereira, and Ricardo Vilaça. MeT: Workload aware elasticity for NoSQL. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 183–196, New York, NY, USA, 2013. ACM.

[122] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*, 2008.

[123] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Trans. Database Syst.*, 38(1):5:1–5:45, April 2013.

[124] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. VLDB Endow.*, 4(8):494–505, May 2011.

[125] Ritendra Datta, Dhiraj Joshi, Jia Li, and James Z. Wang. Image retrieval: Ideas, influences, and trends of the new age. *ACM Computing Surveys*, 40, 2008.

[126] Sebastiaan de Hoogh. *Design of large scale applications of secure multiparty computation: secure linear programming.* PhD thesis, Eindhoven University of Technology, 2012.

[127] Sebastiaan de Hoogh, Berry Schoenmakers, and Meilof Veeningen. Certificate validation in secure computation and its use in verifiable linear programming. In *Progress in Cryptology - AFRICACRYPT 2016 - 8th International Conference on Cryptology in Africa, Fes, Morocco, April 13-15, 2016, Proceedings*, pages 265–284, 2016.

[128] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proc. of the TACAS'08/ETAPS'08*, 2008.

[129] Jeffrey Dean and Luiz Andr Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[130] 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. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[131] Yvo Desmedt. Threshold cryptography. *European Transactions on Telecommunications*, 5(4):449–457, 1994.

[132] T. T. Do, E. Kijak, T. Furon, and L. Amsaleg. Challenging the security of content-based image retrieval systems. In *IEEE International Workshop on Multimedia Signal Processing*, MMSP, 2010.

[133] D. Dobre, G. O. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolic. Powerstore: Proofs of writing for efficient and robust storage. In *Proc. of the CCS*, 2013.

[134] Dan Dobre, Paolo Viotti, and Marko Vukolic. Hybris: Robust hybrid cloud storage. *SoCC*, 2014.

[135] Danny Dolev and H. Raymond Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, November 1983.

[136] Thibault Dory, Boris Mejas, Peter Van Roy, and Nam-Luc Tran. Measuring elasticity for cloud databases. In *Proceedings of the The Second International Conference on Cloud Computing, GRIDs, and Virtualization*, 2011.

[137] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS*, pages 617–624, 2002.

[138] Assia Doudou, Benoit Garbinato, Rachid Guerraoui, and André Schiper. Muteness failure detectors: Specification and implementation. In *Proc. 3rd European Dependable Computing Conference (EDCC-3)*, volume 1667 of *Lecture Notes in Computer Science*, pages 71–87. Springer, 1999.

[139] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A Pras. Inside Dropbox: Understanding personal cloud storage services. In *IMC*, 2012.

[140] E.P. Duarte, T. Garrett, L.C.E. Bona, R. Carmo, and A.P. Zge. Finding stable cliques of planetlab nodes. In *Proc. of the 40th IEEE/IFIP Int. Conference on Dependable Systems and Networks*, 2010.

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

[142] Richard Ekwall and Andr Schiper. Modeling and validating the performance of atomic broadcast algorithms in high latency networks. In *Proc. of Euro-Par*, 2007.

[143] K. El Emam, F. K. Dankar, R. Issa, E. Jonker, D. Amyot, E. Cogo, J. P. Corriveau, M. Walker, S. Chowdhury, R. Vaillancourt, T. Roffey, and J. Bottomley. A globally optimal k-anonymity method for the de-identification of health data. *Journal of the American Medical Informatics Association*, 16(5):670–682, 2009.

[144] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 301–312, New York, NY, USA, 2011. ACM.

[145] Zekeriya Erkin, Martin Franz, Jorge Guajardo, Stefan Katzenbeisser, Inald Lagendijk, and Tomas Toft. Privacy-preserving face recognition. In *The annual Privacy Enhancing Technologies Symposium*, PETS, 2009.

[146] Bernardo Ferreira, João Rodrigues, João Leitão, and Henrique Domingos. Privacy-preserving content-based image retrieval in the cloud. *CoRR*, abs/1411.4862, 2014.

[147] Dario Fiore, Rosario Gennaro, and Valerio Pastro. Efficiently Verifiable Computation on Encrypted Data. In *Proceedings of CCS*, 2014.

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

[149] Sharon Fisher. On the quest for the mysterious source of the "data loss causes company failure" statistic. `http://itknowledgeexchange.techtarget.com/storage-disaster-recovery/on-the-quest-for-the-mysterious-source-of-the-data-loss-causes-company-failure-statistic` February 2014.

[150] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Qian Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: the qbic system. *Computer*, 28, 1995.

[151] A. Folkers and H. Samet. Content-based image retrieval using fourier descriptors on a logo database. In *16th International Conference on Pattern Recognition*, ICPR, 2002.

[152] E. Gafni and L. Lamport. Disk paxos. *Distributed Computing*, 16(1), 2003.

[153] Steven D. Galbraith, Kenneth G. Paterson, and Nigel P. Smart. Pairings for cryptographers. *Discrete Applied Mathematics*, 156(16):3113–3121, 2008.

[154] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO '84*, volume 196 of *LNCS*, pages 10–18. Springer, 1984.

[155] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.*, 30(4):14:1–14:26, November 2012.

[156] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology: Eurocrypt 2015*, volume 9057 of *Lecture Notes in Computer Science*, pages 281–310. Springer, 2015.

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

[158] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers. In *Proceedings of CRYPTO*, 2010.

[159] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic Span Programs and Succinct NIZKs without PCPs. In *Proceedings of EUROCRYPT*. 2013.

[160] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and Fact-Track Multiparty Computations with Applications to Threshold Cryptography. In *Proceedings of PODC*, 1998.

[161] G. Gibson et al. A cost-effective, high-bandwidth storage architecture. In *Proc. of the ASPLOS*, 1998.

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

[163] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in scatter. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles – SOSP'11*, 2011.

[164] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proceedings of STOC*, 2013.

[165] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proc. of the DSN*, 2004.

[166] Google. Google drive. `https://drive.google.com/`, 2016.

[167] S.Dov Gordon, Jonathan Katz, Feng-Hao Liu, Elaine Shi, and Hong-Sheng Zhou. Multi-client verifiable computation with stronger security guarantees. In Yevgeniy Dodis and JesperBuus Nielsen, editors, *Theory of Cryptography*, volume 9015 of *Lecture Notes in Computer Science*, pages 144–168. Springer Berlin Heidelberg, 2015.

[168] Jens Groth. Short Pairing-Based Non-interactive Zero-Knowledge Arguments. In *Proceedings of ASIACRYPT*, 2010.

[169] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: Replication at the speed of multi-core. In *Proc. of the 9th European Conference on Computer Systems – EuroSys '14*, 2014.

[170] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*. ACM Press & Addison-Wesley, New York, 1993.

[171] J. Hamilton. Observations on errors, corrections, and trust of dependent systems. `http://goo.gl/LPTJoO`, 2012.

[172] Seungyeop Han, Haichen Shen, Taesoo Kim, Arvind Krishnamurthy, Thomas Anderson, and David Wetherall. MetaSync: File synchronization across multiple untrusted storage services. In *Proc. of the 2015 USENIX ATC*, 2015.

[173] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead Byzantine fault-tolerant storage. In *Proc. of the SOSP*, 2007.

[174] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1), 1991.

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

[176] Pedro Hernandez. Small business IT survey: No backup, no data, no business. `http://www.smallbusinesscomputing.com/biztools/small-business-it-survey-no-backup-no-data-no-business.html`, May 2014.

[177] B. Hou, F. Chen, Z. Ou, R. Wang, and M. Mesnier. Understanding I/O performance behaviors of cloud storage from a client's perspective. *MSST*, 2016.

[178] C. Y. Hsu, C. S. Lu, and S. C. Pei. Image feature extraction in encrypted domain with privacy-preserving sift. *IEEE Transactions on Image Processing*, 21, 2012.

[179] Chao-Yung Hsu, Chun-Shien Lu, and Soo-Chang Pei. Secure and robust sift. In *ACM International Conference on Multimedia*, MM, 2009.

[180] Chao-Yung Hsu, Chun-Shien Lu, and Soo-Chang Pei. Homomorphic encryption-based secure sift for privacy-preserving feature extraction. In *Media Forensics and Security part of the IS&T-SPIE Electronic Imaging Symposium*, SPIE, 2011.

[181] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale services. In *Proc. of the USENIX Annual Technical Conference*, 2010.

[182] J. Illingworth, J. Kittler, and J. Princen. *Shape Detection in Computer Vision Using the Hough Transform*. Springer-Verlag New York, Inc., 1 edition, 1988.

[183] Tibor Jager. Verifiable random functions from weaker assumptions. In *Proc. 12th Theory of Cryptography Conference (TCC 2015)*, volume 9015 of *Lecture Notes in Computer Science*, pages 121–143. Springer, 2015.

[184] Thomas P. Jakobsen, Jesper Buus Nielsen, and Claudio Orlandi. A framework for outsourcing of secure computation. In *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security, CCSW '14, Scottsdale, Arizona, USA, November 7, 2014*, pages 81–92, 2014.

[185] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, 45(3), 1998.

[186] Minwen Ji, Alistair C Veitch, John Wilkes, et al. Seneca: remote mirroring done write. In *Proceedings of the 2003 USENIX Annual Technical Conference (ATC'03)*, 2003.

[187] Pascal Junod and Alexandre Karlov. An efficient public-key attribute-based broadcast encryption scheme allowing arbitrary access policies. In *ACM Workshop on Digital Rights Management*, pages 13–24. ACM Press, 2010.

[188] Flavio Junqueira, Yanhua Mao, and Keith Marzullo. Classic Paxos vs Fast Paxos: Caveat emptor. In *Proc. of the Workshop on Hot Topics in System Dependability*, 2007.

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

[190] Seny Kamara, Payman Mohassel, and Ben Riva. Salus: a system for server-aided secure function evaluation. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 797–808, 2012.

[191] 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 *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 295–308, New York, NY, USA, 2012. ACM.

[192] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: Execute-verify replication for multi-core servers. In *Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation – OSDI'12*, 2012.

[193] Yan Ke and R. Sukthankar. Pca-sift: a more distinctive representation for local image descriptors. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, CVPR, 2004.

[194] Kimberly Keeton, Cipriano A Santos, Dirk Beyer, Jeffrey S Chase, and John Wilkes. Designing for disasters. *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)*, 2004.

[195] Bettina Kemme, Ricardo J. Peris, and Marta Patio-Martnez. *Database Replication*. Morgan & Claypool, 2010.

[196] Ioannis Konstantinou, Evangelos Angelou, Christina Boumpouka, Dimitrios Tsoumakos, and Nectarios Koziris. On the elasticity of NoSQL databases over cloud management platforms. In *Proc. of the 20th ACM international conference on Information and knowledge management – CIKM '11*, 2011.

[197] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, January 2010.

[198] Ramakrishna Kotla and Mike Dahlin. High throughput byzantine fault tolerance. In *Proc. of the 2004 International Conference on Dependable Systems and Networks – DSN'04*, 2004.

[199] Edward Kovacs. Downtime and data loss cost enterprises $1.7 trillion per year: EMC. `http://www.securityweek.com/downtime-and-data-loss-cost-enterprises-17-trillion-year-emc`, December 2014.

[200] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: multi-data center consistency. In *Eighth Eurosys Conference 2013*, pages 113–126, 2013.

[201] Hugo Krawczyk. Secret sharing made short. In *Proc. of the CRYPTO*, 1993.

[202] Andrew Krioukov, Prashanth Mohan, Sara Alspaugh, Laura Keys, David Culler, and Randy Katz. Napsac: Design and implementation of a power-proportional web cluster. In *Proc. of the 1st ACM Workshop on Green Networking*, 2010.

[203] Kripa Krishnan. Weathering the unexpected. *Commun. ACM*, 55:48–52, November 2012.

[204] John Kubiatowicz *et al.* OceanStore: An architecture for global-scale persistent storage. In *ASPLOS*, 2000.

[205] Petr Kuznetsov and Rodrigo Rodrigues. BFTW3: Why? When? Where? Workshop on the theory and practice of Byzantine fault tolerance. *SIGACT News*, 40(4):82–86, January 2010.

[206] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, November 1992.

[207] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[208] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[209] Leslie Lamport. On interprocess communication (part II). *Distributed Computing*, 1(1), 1986.

[210] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.

[211] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[212] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, July 1982.

[213] A. Lathey, P. K. Atrey, and N. Joshi. Homomorphic low pass filtering on encrypted multimedia over cloud. In *IEEE International Conference on Semantic Computing*, ICSC, 2013.

[214] Ankita Lathey and Pradeep K. Atrey. Image enhancement in encrypted domain over cloud. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 11, 2015.

[215] William LeFebvre. Cnn. com: Facing a world crisis. In *LISA*, 2001.

[216] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 1–14. USENIX Association, 2009.

[217] Michael S. Lew, Nicu Sebe, Chabane Djeraba, and Ramesh Jain. Content-based multimedia information retrieval: State of the art and challenges. *ACM Transactions on Multimedia Computing, Communications, and Applications*, 2, 2006.

[218] Barbara Liskov. From viewstamped replication to Byzantine fault tolerance. In Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors, *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*, pages 121–149. Springer, 2010.

[219] Jacob Lorch, Atul Adya, William Bolosky, Ronnie Chaiken, John Douceur, and Jon Howell. The SMART way to migrate replicated stateful services. In *Proceedings of the 1st ACM European Systems Conference*, October 2006.

[220] Rafael Los, Dave Shacklenford, and Bryan Sullivan. The notorious nine: Cloud Computing Top Threats in 2013. Technical report, Cloud Security Alliance (CSA), February 2013.

[221] D. G. Lowe. Object recognition from local scale-invariant features. In *IEEE International Conference on Computer Vision*, ICCV, 1999.

[222] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60, 2004.

[223] W. Lu, A. L. Varna, A. Swaminathan, and M. Wu. Secure image retrieval through feature protection. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP, 2009.

[224] Wenjun Lu, Ashwin Swaminathan, Avinash L. Varna, and Min Wu. Enabling search over encrypted multimedia databases. In *Media Forensics and Security part of the IS&T-SPIE Electronic Imaging Symposium*, SPIE, 2009.

[225] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kauffman, 1996.

[226] Anna Lysyanskaya. Unique signatures and verifiable random functions from the DH-DDH separation. In *Advances in Cryptology: CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 597–612. Springer, 2002.

[227] D. Malkhi and M.K. Reiter. Secure and scalable replication in Phalanx. In *Proc. of the SRDS*, 1998.

[228] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4), 1998.

[229] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for WANs. In *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[230] Joo Marques-Silva and Karem Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. on Computers*, 48:506–521, 1999.

[231] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal Byzantine storage. In *Proc. of the DISC*, 2002.

[232] Babu M. Mehtre, Mohan S. Kankanhalli, and Wing Foon Lee. Shape measures for content based image retrieval: A comparison. *Information Processing and Management*, 33, 1997.

[233] Ricardo Mendes, Tiago Oliveira, Vinicius Cogo, Nuno Neves, and Alysson Bessani. Charon: A Dependable Cloud-of-Clouds System for Storing and Sharing Big Data. *Under submission*, 2016.

[234] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 120–130, 1999.

[235] Microsoft. Microsoft onedrive. `https://onedrive.live.com/about/pt-br/`, 2016.

[236] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2016.

[237] Umar Farooq Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. RemusDB: Transparent high availability for database systems. *The VLDB Journal*, 22(1), 2013.

[238] Shigeo Mitsunari. A Fast Implementation of the Optimal Ate Pairing over BN curve on Intel Haswell Processor. Cryptology ePrint Archive, Report 2013/362, 2013. `http://eprint.iacr.org/`.

[239] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1), 1992.

[240] Payman Mohassel and Matthew K. Franklin. Efficiency Tradeoffs for Malicious Two-Party Computation. In *Proceedings of PKC*, 2006.

[241] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proc. of 24th ACM Symposium on Operating Systems Principles*, 2013.

[242] Jim Mutch and David G. Lowe. Object class recognition and localization using sparse features with limited receptive fields. *International Journal of Computer Vision*, 80, 2008.

[243] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009.

[244] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony Rowstron. Everest: Scaling down peak loads through i/o off-loading. In *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation – OSDI'08*, 2008.

[245] Nasuni. Nasuni UniFS. `http://www.nasuni.com/`, 2016.

[246] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems – EuroSys '10*, pages 237–250, 2010.

[247] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at Facebook. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation*, April 2013.

[248] Brian M. Oki and Barbara Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 8–17, 1988.

[249] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[250] Diego Ongaro and John Ousterhout. In search for an understandable consensus algorithm. In *Proc. of the USENIX Annual Technical Conference – USENIX ATC 2014*, June 2014.

[251] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–319, 2014.

[252] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich. Scifi - a system for secure face identification. In *IEEE Symposium on Security and Privacy*, S&P, 2010.

[253] Ricardo Padilha and Fernando Pedone. Augustus: Scalable and robust storage for cloud applications. In *Proceedings of the eigth conference on Computer systems*, EuroSys '13, 2013.

[254] Panzura. Panzura CloudFS. `http://panzura.com/`, 2016.

[255] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *Proceedings of S&P*, 2013.

[256] Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. Snapmirror: file system based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.

[257] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[258] F. Pedone, C. E. Bezerra, and R. van Renesse. Scalable state-machine replication. In *44th International Conference on Dependable Systems and Networks (DSN 2014)*, 2014.

[259] Andreas Peter, Erik Tews, and Stefan Katzenbeisser. Efficiently outsourcing multiparty computation under multiple keys. *IEEE Transactions on Information Forensics and Security*, 8(12):2046–2058, 2013.

[260] Duong Hieu Phan, David Pointcheval, and Viet Cuong Trinh. Multi-Channel Broadcast Encryption. In *Proceedings of the 8th ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS '13)*, ACM Press, 2013.

[261] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. Visigoth fault tolerance. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 8:1–8:14, New York, NY, USA, 2015. ACM.

[262] Jehan Pris. Voting with witnesses: A consistency scheme for replicated files. In *In Proceedings of the 6th International Conference on Distributed Computing Systems*, 1986.

[263] Hyperledger project. Fabric. http://github.com/hyperledger/fabric, 2016.

[264] Krishna P. N. Puttaswamy, Thyaga Nandagopal, and Murali Kodialam. Frugal storage for cloud file systems. In *Proc. of the 10th ACM European Systems Conference – EuroSys'12*, 2012.

[265] Zhan Qin, Jingbo Yan, Kui Ren, Chang Wen Chen, and Cong Wang. Towards efficient privacy-preserving image feature extraction in cloud computing. In *22Nd ACM International Conference on Multimedia*, MM, 2014.

[266] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2), 1989.

[267] Shriram Rajagopalan, Brendan Cully, Ryan O'Connor, and Andrew Warfield. SecondSite: disaster tolerance as a service. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE'12)*, 2012.

[268] Jun Rao, Eugene J. Shenkita, and Sandeep Tata. Using Paxos to build a scalable, consistent, and highly available datastore. *The VLDB Journal*, 4(4), 2011.

[269] Michael K. Reiter and Kenneth P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, 16(3):986–1009, May 1994.

[270] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[271] Glen Robinson, Attila Narin, and Chris Elleman. Using amazon web services for disaster recovery. Amazon Web Services white paper, December 2014.

[272] S. Roy and Q. Sun. Robust hash for detecting and localizing image tampering. In *2007 IEEE International Conference on Image Processing*, ICIP, 2007.

[273] Yong Rui and Thomas S. Huang. Image retrieval: Current techniques, promising directions and open issues. *Journal of Visual Communication and Image Representation*, 10, 1999.

[274] Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Efficient privacy-preserving face recognition. In *12th International Conference on Information Security and Cryptology*, ICISC, 2010.

[275] P. Samarati. Protecting respondents' identities in microdata release. *IEEE Trans. on Knowl. and Data Eng.*, 13(6):1010–1027, November 2001.

[276] N. Santos and A. Schiper. Achieving high-throughput state machine replication in multi-core systems. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 266–275, July 2013.

[277] N. Schiper, P. Sutra, and F. Pedone. Genuine versus non-genuine atomic multicast protocols for wide area networks: An empirical study. In *Proc. of the 28th IEEE Int. Symposium on Reliable Distributed Systems*, 2009.

[278] Fred Schneider. Implementing fault-tolerant service using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[279] Berry Schoenmakers, Meilof Veeningen, , and Niels de Vreede. Trinocchio: Privacy-friendly outsourcing by distributed verifiable computation. Cryptology ePrint Archive, Report 2015/480, 2015. `http://eprint.iacr.org/2015/480`.

[280] Berry Schoenmakers and Meilof Veeningen. Universally Verifiable Multiparty Computation from Threshold Homomorphic Cryptosystems. In *Proceedings of ACNS*, 2015. `http://eprint.iacr.org/2015/058`.

[281] Berry Schoenmakers, Meilof Veeningen, and Niels de Vreede. Trinocchio: Privacy-preserving outsourcing by distributed verifiable computation. In *Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings*, pages 346–366, 2016.

[282] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *Proc. VLDB Endow.*, 7(12):1035–1046, August 2014.

[283] J. Shashank, P. Kowshik, K. Srinathan, and C. V. Jawahar. Private content based image retrieval. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, CVPR, 2008.

[284] Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio P. Junqueira. Dynamic reconfiguration of primary/backup clusters. In *Proceedings of the USENIX Annual Technical Conference – ATC'12*, 2012.

[285] Arnold W. M. Smeulders, Marcel Worring, Simone Santini, Amarnath Gupta, and Ramesh Jain. Content-based image retrieval at the end of the early years. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, TPAMI, 2000.

[286] Susan Snedaker. *Business continuity and disaster recovery planning for IT professionals*. Newnes, 2013.

[287] J. Sousa and A. Bessani. Separating the WHEAT from the chaff: An empirical design for geo-replicated state machines (extended version). Technical Report TR 2015-04, Department of Informatics, Faculty of Sciences of the University of Lisboa, July 2015.

[288] Joo Sousa and Alysson Bessani. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *Proc. of the 9th European Dependable Computing Conference*, 2012.

[289] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[290] Michael Stonebraker and Lawrence A Rowe. The design of postgres. In *Proceedings of the 1986 ACM SIGMOD ACM SIGMOD international conference on Management of data*, 1986.

[291] Jeremy Stribling *et al.* Flexible, wide-area storage for distributed system with WheelFS. In *NSDI*, 2009.

[292] Tim Swanson. Consensus-as-a-service: A brief report on the emergence of permissioned, distributed ledger systems. Report, available online, April 2015.

[293] Latanya Sweeney. K-anonymity: A model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):557–570, October 2002.

[294] Symantec. SMB (Small and Medium Business) security and data protection: survey shows high concern, less action. White paper: SMB Survey. Available at `http://eval.symantec.com/mktginfo/enterprise/other_resources/b-SMB-Protection-Gap_WP_20094842.en-us.pdf`, 2009.

[295] Symantec. Ransomware and business 2016. ISTR Speacial Report. Available at `http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/ISTR2016_Ransomware_and_Businesses.pdf`, 2016.

[296] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3):245–256, November 2014.

[297] Haowen Tang, Fangming Liu, Guobin Shen, Yuchen Jin, and Chuanxiong Guo. UniDrive: Synergize multiple consumer cloud storage services. In *Proc. of the ACM/IFIP/USENIX Middleware'15*, 2015.

[298] Qiang Tang. Type-based proxy re-encryption and its construction. In *INDOCRYPT 2008*, volume 5365 of *Lecture Notes in Computer Science*, pages 130–144. Springer, 2008.

[299] Vasily Tarasov, Abhishek Gupta, Kumar Sourav, Sagar Trehan, and Erez Zadok. Terra incognita: On the practicality of user-space file systems. In *Proceedings of the 7th USENIX workshop on hot topics in Storage and File Systems (HotStorage'15)*, 2015.

[300] Philip M. Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *Seventh Symposium on Reliable Distributed Systems, SRDS 1988, Columbus, Ohio, USA, October 10-12, 1988, Proceedings*, pages 93–100, 1988.

[301] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The SCADS Director: Scaling a distributed storage system under stringent performance requirements. In *Proc. of the 9th USENIX Conference on File and Stroage Technologies – FAST'11*, 2011.

[302] M. Upmanyu, A. M. Namboodiri, K. Srinathan, and C. V. Jawahar. Efficient privacy preserving video surveillance. In *IEEE International Conference on Computer Vision*, ICCV, 2009.

[303] Joost van de Weijer and Cordelia Schmid. Coloring local feature extraction. In *9th European Conference on Computer Vision*, ECCV, 2006.

[304] G. Veronese, M. Correia, A.N. Bessani, and Lau Cheuk Lung. EBAWA: Efficient Byzantine agreement for wide-area networks. In *Proc. of the 12th IEEE Int. High Assurance Systems Engineering Symposium*, 2010.

[305] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. Efficient Byzantine fault-tolerance. *IEEE Trans. Computers*, 62(1):16–30, 2013.

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

[307] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. BlueSky: A cloud-backed file system for the enterprise. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, 2012.

[308] Marko Vukolic. SUPERCLOUD, D3.1 - Architecture for Data Management, 2015.

[309] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *Open Problems in Network Security, Proc. IFIP WG 11.4 Workshop (iNetSec 2015)*, volume 9591 of *Lecture Notes in Computer Science*, pages 112–125. Springer, 2016.

[310] Michael Walfish and Andrew J. Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2), February 2015.

[311] Q. Wang, S. Hu, J. Wang, and K. Ren. Secure surfing: Privacy-preserving speeded-up robust feature extractor. In *IEEE International Conference on Distributed Computing Systems*, ICDCS, 2016.

[312] Shumiao Wang, Mohamed Nassar, Mikhail Atallah, and Qutaibah Malluhi. Secure and private outsourcing of shape-based feature extraction. In *15th International Conference on Information and Communications Security*, ICICS, 2013.

[313] 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*, June 2012.

[314] 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 *OSDI*, 2006.

[315] L. Weng, L. Amsaleg, A. Morton, and S. Marchand-Maillet. A privacy-preserving framework for large-scale content-based information retrieval. *IEEE Transactions on Information Forensics and Security*, 10, 2015.

[316] Timothy Wood, Emmanuel Cecchet, KK Ramakrishnan, Prashant Shenoy, Jacobus Van Der Merwe, and Arun Venkataramani. Disaster recovery as a cloud service: Economic benefits & deployment challenges. In *Proceedings of the 1st USENIX workshop on hot topics in cloud computing (HotCloud'10)*, 2010.

[317] Timothy Wood, H Andrés Lagar-Cavilla, KK Ramakrishnan, Prashant Shenoy, and Jacobus Van der Merwe. Pipecloud: using causality to overcome speed-of-light delays in cloud-based disaster recovery. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC'11)*, 2011.

[318] Y. Ye, L. Xiao, I-L. Yen, and F. Bastani. Secure, dependable, and high performance cloud storage. In *Proc. of the SRDS*, 2010.

[319] X. Yuan, X. Wang, C. Wang, A. Squicciarini, and K. Ren. Enabling privacy-preserving image-centric social discovery. In *IEEE International Conference on Distributed Computing Systems*, ICDCS, 2014.

[320] X. Zhang and H. Cheng. Histogram-based retrieval for encrypted jpeg images. In *IEEE China Summit International Conference on Signal and Information Processing*, ChinaSIP, 2014.

[321] L. Zheng and S. Wang. Visual phraselet: Refining spatial constraints for large scale image search. *IEEE Signal Processing Letters*, 20, 2013.

[322] Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, 2002.

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