UC Santa Barbara
UC Santa Barbara Electronic Theses and Dissertations

Title
Enhancing the performance, fault tolerance, and security of distributed data management systems

Permalink
https://escholarship.org/uc/item/4zx2m54f

Author
Maiyya, Sujaya Anantha

Publication Date
2022

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

University of California
Santa Barbara

Enhancing the performance, fault tolerance, and security of distributed data management systems

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Sujaya Maiyya

Committee in charge:

Professor Divyakant Agrawal, Co-Chair
Professor Amr El Abbadi, Co-Chair
Professor Prabhanjan Ananth

June 2022

The Dissertation of Sujaya Maiyya is approved.

Professor Prabhanjan Ananth

Professor Divyakant Agrawal, Committee Co-Chair

Professor Amr El Abbadi, Committee Co-Chair

June 2022

Enhancing the performance, fault tolerance, and security of distributed data management systems

Copyright © 2022

by

Sujaya Maiyya

To my family – Jim, Appa Amma, Shunti & Shenga

Acknowledgements

"None of us, acting alone, can achieve the success" - Nelson Mandela

The completion of this dissertation, and hence my PhD, was not a solo act. Many people have played a key role in guiding me in the right direction and proving a steadfast support during this journey. I want to acknowledge these individuals and convey my gratitude.

My sincere thanks go to my advisors Divy Agrawal and Amr El Abbadi. Their constant support and encouragement has been paramount in all my accomplishments. Back in 2016-17, I was eager to graduate with a master's degree as quickly as possible; it was Amr's Distributed Systems class that convinced me to stay and pursue a master's project with him. While working on my master's project, Amr and Divy's passion for distributed systems and databases enthralled me and their friendly and supportive student advising techniques encouraged me to continue on to pursue my PhD with them. Their training over the years has instilled courage in me to take on open and challenging research ideas and taught me to think critically about open research problems. With regard to teaching, I have worked as a teaching assistant to both Amr and Divy. Amr, by setting an example, has shown me the joy of teaching and he has helped me push my boundaries in preparing and delivering lectures. One of the things, in my humble opinion, that makes Amr and Divy standout is their encouragement to allow their students to pursue non-academic endeavors such as volunteering as a graduate student representative within the CS department at UCSB or to serve in Diversity & Inclusion committees in the database community. Although these endeavors consume our research time, Amr and Divy recognize the significance of taking on such roles in our overall development and hence they are nothing but encouraging in this regard. All in all, I am and will continue to be forever grateful for their training in research, teaching, and service, which will aid me tremendously in the academic career I am about embark upon.

I am also deeply grateful to Prabhanjan for being my committee member and for educating me many times over about complex topics in cryptography. I fondly recall the time Prabhanjan spent during his second interview visit to UCSB where I bombarded him with my ideas and questions regarding an ongoing project; I am grateful for his suggestions that helped solidify our paper. I am also indebted to professor and current department chair Tevfik Bultan for writing many letters of recommendation over the years for various fellowships and academic jobs. Tevfik's positive emails during the challenging times of the pandemic helped me, and I am sure many others, not lose my hope; our department could not have asked for a better chair during those challenging times. I am also thankful to Olivier Tardieu, my mentor during my internship at IBM Research. Olivier's kind and supportive nature not only helped me during my internship but also during my academic job search where he provided recommendation letters.

I am grateful to Professors Rachel Lin and Stefano Tessaro for many hours of fascinating discussions on ORAM; it is through these discussions that I understand systems security better. I extend my thanks to Professor Ambuj Singh for always identifying and believing in the inner academic in me, even before I recognized it myself. I am thankful for Professor Jonathan Balkind for being a mentor-figure during my academic interview process and providing me with great tools and tactics to navigate the process. A hearty thanks to Professor Scott Marcus from the music department for teaching me how to play sitar, which helped me retain a sense of work-life balance. I thank professors Ömer Eğecioğlu, Trinabh Gupta, Chandra Krintz, Daniel Lokshtanov, Tim Sherwood, Rich Wolski, Xifeng Yan, and Ben Zhao for their classes, technical discussions, or general advice.

I have been fortunate to have had wonderfully talented and gregarious lab mates - Faisal, Vaibhav, Victor, Mohammad, Ishtiyaque, Fuheng, and Lawrence. I have had an immense joy discussing technical problems with them and have realized what a great

conference-travel partner Victor is. Faisal has been a great mentor, providing me feedbacks on my talks and general academic advice over the years. I also thank Danny, Hari, Aarti, Seif, Yuval, Sharath, and Daniel for providing me with the opportunity to mentor them for their undergraduate or master's projects. I am also grateful for all the behind-the-scenes support provided by the CS department staff Greta, Maritza, Karen, Samantha, Nicole, and Benji.

The duration of my graduate school would not have been as much fun without my friends whose memories I will cherish forever. My special thanks to my friends Devin and Aishwarya, one for convincing me to pursue a PhD and one for helping me sustain it, and to Rachel for becoming a pseudo-DSLer for me. My many thanks to my friends for sharing their time with me: Angela, Arnab, Atefeh, Bradley, Burak, Cliff, Connor, Deeksha, Fatih, Goksu, Haraldur, Lale, Mohith, Neeraj, Nevena, Nimisha, Omid, Ryan, Seemanta, and Shashank.

And last but not the least, I could not have completed my PhD without the constant support of my family. I am grateful to my sister, Sindhu, my aunts, uncles, and cousins for their positive encouragement, and to my in-laws Alice, Glenn, and Danny for being my wonderful family outside of my home country. I am fortunate to have the parents I have and I am indebted to them for teaching me to believe in myself time and again and guiding me to never be afraid of trying new things. And finally, I am extremely grateful to Jim for his thoughtfulness and patience, for being a constant support at my lowest and highest, for patiently proofreading all my statements, for caring for our cats Shunti & Shenga, and for providing me the space to grow in my career.

Curriculum Vitæ
Sujaya Maiyya

Education

2022	Ph.D. in Computer Science (Expected), University of California, Santa Barbara.
2017	M.Sc. in Computer Science, University of California, Santa Barbara.
2014	B.E in Information Science, PES Institute of Technology, Bangalore.

Publications

[1] **Sujaya Maiyya**, Y. Steinhart, P. Ananth, D. Agrawal, and A. El Abbadi. *ORTOA: One Round Trip Oblivious Access.* In submission [145].

[2] MJ. Amiri, D. Shu, **Sujaya Maiyya**, D. Agrawal, and A. El Abbadi. Ziziphus: Scalable Data Management Across Byzantine Edge Servers. In submission [9].

[3] **Sujaya Maiyya**, S. Ibrahim, C. Scarberry, D. Agrawal, A. El Abbadi, H. Lin, S. Tessaro, and V. Zakhary. *QuORAM: A Quorum-based Replicated ORAM Datastore.* To appear in USENIX Security 2022 [147].

[4] **Sujaya Maiyya**, I. Ahmad, D. Agrawal, and A. El Abbadi. *Samya: Geo-distributed data system for high contention data aggregates.* International Conference on Data Engineering (ICDE), 2021 [144].

[5] F. Zhao, **Sujaya Maiyya**, R. Wiener, D. Agrawal, and A. El Abbadi. *Kll±: Approximate quantile sketches over dynamic datasets.* Proceedings of the VLDB Endowment, 2021 [229].

[6] **Sujaya Maiyya**, DBH. Cho, D. Agrawal, and A. El Abbadi. *Fides: Managing data on untrusted infrastructure.* International Conference on Distributed Computing Systems (ICDCS), 2020 [146].

[7] MJ. Amiri, **Sujaya Maiyya**, D. Agrawal, and A. El Abbadi. *Seemore: A fault-tolerant protocol for hybrid cloud environments.* International Conference on Data Engineering (ICDE), 2020 [8].

[8] **Sujaya Maiyya**, V. Zakhary, MJ. Amiri, D. Agrawal, and A. El Abbadi. *Database and distributed computing foundations of blockchains.* (Tutorial) In SIGMOD, 2019 [150].

[9] **Sujaya Maiyya**, F. Nawab, D. Agrawal, and A. El Abbadi. *Unifying consensus and atomic commitment for effective cloud data management.* Proceedings of the VLDB Endowment, 12(5):611–623, 2019 [148].

[10] V. Zakhary, MJ Amiri, **Sujaya Maiyya**, D. Agrawal, and A. El Abbadi. *Towards global asset management in blockchain systems.* BCDL co-located with VLDB, 2019 [222].

[11] **Sujaya Maiyya**, V. Zakhary, D. Agrawal, and A. El Abbadi. *Database and distributed computing fundamentals for scalable, fault-tolerant, and consistent maintenance of blockchains.* (Tutorial) Proceedings of the VLDB Endowment, 2018 [149].

[12] V. Arora, RKS. Babu, **Sujaya Maiyya**, D. Agrawal, A. El Abbadi, X. Xue, et al. *Dynamic timestamp allocation for reducing transaction aborts.* In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), 2018 [13].

Please Note: Text and figures from these papers appear in this dissertation.

Abstract

Enhancing the performance, fault tolerance, and security of distributed data management systems

by

Sujaya Maiyya

Individuals and enterprises produce over 2.5 exabytes (10^{18} bytes) of data everyday. Much of this data - including sensitive and private information - is stored with and managed by third-parties, such as Amazon Web Services or Google Cloud. These companies can lose millions to billions of dollars in sales if their data access latencies increase by only a few hundred milliseconds. Achieving data fault tolerance – a necessary primitive of database systems – while maintaining low access latency is particularly challenging. Hence, reducing data access latency to improve performance and guaranteeing data fault tolerance received the highest priority while designing cloud data management systems. But the ever growing number and sophistication of cyber attacks on the cloud coupled with increasing legal requirements for data privacy and security (e.g., GDPR or HIPAA) have forced cloud providers to re-evaluate their priorities. However, there exists a fundamental trade-off between security and efficiency in data management systems.

This dissertation discusses designing and evaluating data management protocols that strike a balance between efficiency, fault tolerance, and security in both trusted and untrusted environments. Before being able to solve security challenges in database systems, we first delve into traditional cloud settings, which assumes trust, to understand existing system designs. Existing cloud databases *replicate* their data to provide fault tolerance and *shard* (or partition) the data and store the shards on multiple servers to provide scalability. In trusted environments, we propose two solutions: *G-PAC*, an atomic com-

mitment protocol that commits transactions accessing data that is both sharded and replicated, and *Samya*, a data system that maintains aggregate data and supports high-contention write-intensive workloads. As the next step towards building secure data systems, to better understand the interplay between multiple security guarantees and performance, we study various blockchain systems – an ideal example where untrusted geo-distributed entities manage critical data.

Equipped with blockchain techniques that protect data, we build three solutions that focus on data *Confidentiality, Integrity,* and *Availability*, more popularly known as the *CIA* triad, which forms the pillars of secure systems. For confidentiality, this dissertation proposes *ORTOA*: a protocol that allows users to read or write data onto an untrusted external server without revealing the type of operation *in a single round*, whereas all existing solutions to hide the type of operation require two rounds of communication. For integrity, this dissertation presents *Fides*: a transactional database system that guarantees data integrity and provides verifiable ACID guarantees. In this work, we also propose *TFCommit* - the first distributed transaction commitment protocol that tolerates up to $n-1$ maliciously failing servers (out of n servers) without using expensive data replication. And for availability, we propose *QuORAM*: the first fully fault-tolerant Oblivious RAM datastore that guarantees data privacy by hiding access patterns of users along with the contents of data.

Contents

Curriculum Vitae **viii**

Abstract **x**

1 Introduction **1**
1.1 Motivation 1
1.2 Dissertation Overview 4
1.3 Dissertation Organization 10

2 Unifying Consensus and Atomic Commitment for Effective Cloud Data Management **11**
2.1 Overview 11
2.2 Introduction 12
2.3 Background 15
2.4 Unifying Consensus and Commitment 20
2.5 Sharding-Only in the Cloud 26
2.6 Replication-Only in the Cloud 31
2.7 Sharding + Replication in the Cloud 33
2.8 Safety in the C&C framework 37
2.9 Evaluation 39
2.10 Related Work 48
2.11 Conclusion 51

3 Samya: Geo-Distributed Data System for High Contention Data Aggregates **52**
3.1 Overview 52
3.2 Introduction 53
3.3 Related Work 57
3.4 Samya Architecture 61
3.5 Samya 63
3.6 Experimental Evaluation 82

3.7 Conclusion . . . 98

4 Fides: Managing Data on Untrusted Infrastructure 100
4.1 Overview . . . 100
4.2 Introduction . . . 101
4.3 Cryptographic Preliminaries . . . 104
4.4 Fides Architecture . . . 107
4.5 Fides . . . 110
4.6 Failure Examples . . . 131
4.7 Evaluation . . . 133
4.8 Related Work . . . 138
4.9 Conclusion . . . 141

5 QuORAM: A Quorum-Replicated Fault Tolerant ORAM Datastore 142
5.1 Overview . . . 142
5.2 Introduction . . . 143
5.3 Background . . . 147
5.4 System and Failure Model . . . 148
5.5 Security Model: Obliviousness in a Replicated ORAM Setting . . . 151
5.6 QuORAM: a replicated ORAM datastore . . . 154
5.7 Evaluation . . . 171
5.8 Security of replicated ORAM datastores . . . 183
5.9 Linearizability . . . 186
5.10 Space analysis . . . 190
5.11 Related Work . . . 193
5.12 Conclusion . . . 195

6 ORTOA: One Round Trip Oblivious Access 196
6.1 Introduction . . . 197
6.2 System and Security Model . . . 200
6.3 FHE based solution . . . 202
6.4 ORTOA . . . 207
6.5 Optimizations . . . 215
6.6 Protocol evaluation . . . 220
6.7 Security of ORTOA . . . 228
6.8 Conclusion . . . 234

7 Concluding remarks 235

8 Future directions 240
8.1 Frequency Smoothing using a BST Framework . . . 240
8.2 High functionality oblivious datastores . . . 242

Chapter 1

Introduction

1.1 Motivation

Enterprises and individuals produced 79 zettabytes (10^{21}) of data in the year 2021. Much of this data is stored in the cloud, typically managed by third-party vendors such as Amazon Web Services [4], Microsoft Azure [153], or Google Cloud [81]. Some of the most common reasons why enterprises migrate to cloud are to reduce deployment costs due to not owning and maintaining on-premise infrastructure, for the scalability and elasticity of the cloud with practically unbounded amount of resources, and for the reliability and global availability of cloud-hosted applications where typically the data is available 99% of the time and clients across the globe can access the data [203]. Although these are irrefutable advantages of why a client should migrate to cloud, there exists many challenges in designing cloud-based data management systems that provide guarantees such as such as providing high performance, scalability and reliability, and security and privacy. In this dissertation, we address three such challenges faced by the cloud provider in developing cloud-based data management systems: *high performance & low latency*, *fault tolerance*, and *security & privacy*.

A recent study by Microsoft shows that people's attention span is now around 8 seconds [156] and, in fact, when interacting with computers, people require even lower latency of less than one second [169, 207]. Companies such as Google and Amazon lose millions to billions of dollars in sales if their data access latencies increase by only a few hundred milliseconds [103]. Hence, in both industry [52, 121, 180, 159, 38] and academia [108, 143, 200, 94, 69, 137], much effort is spent on designing data management systems that minimize access latency. But designing data management systems that provide low latency for *all* clients becomes challenging when the clients are geo-distributed. For example: if a database system chooses to store all its data in a US-West datacenter, a client close to the US-West datacenter may have low data access latency but a client from Asia will inevitably face large data access latency. Therefore, achieving low latency, which directly impacts performance, becomes one of the main challenges in designing database systems for global-scale applications.

Another major challenge faced by cloud-based database designers is *fault tolerance*. Due to reduced deployment and maintenance costs, much of the cloud architecture today relies on hundreds of thousands of commodity servers rather than sophisticated mainframe computers [12, 17, 101]. Although individual commodity hardware can last between 3-5 years with minimal failures, at the scale of hundreds of thousands of machines, cloud datacenters face constant server failures [18]. Many works have studied the unavoidable nature of individual servers, server racks, and even the entire datacenter-level failures [210, 64, 24, 212, 93]. Given this inevitability, database systems must ensure that a hardware failure *does not* result in data losses, i.e., ensure that an application's data is fault-tolerant even in the presence of hardware failures. Modern day databases typically ensure this by *replicating* the data across different geographically distant datacenters [31, 188, 45, 198]. However, replicating data presents further challenges in guaranteeing the correctness of the data: *when a client updates data in one datacenter, should all copies*

of the data be updated immediately or can an application allow other users to access stale versions of the data from un-updated replicas? If an application requires propagating the update immediately to all replicas, this further aggravates the latency issue, since updating cross-datacenter replicas typically requires hundreds of milliseconds. Hence, achieving fault tolerance while maintaining low latency and high performance becomes one of the other major difficulty in designing data management systems.

Finally, apart from achieving low latency and fault tolerance, another major challenge in designing data management systems is *security & privacy*. With the ever increasing adaption of cloud where individuals and enterprises outsource their data storage and management to external cloud providers, the clients lose control over how the external cloud views and manages their data internally. The growing number and sophistication of cyber attacks on the cloud [104] have attracted more attention to the security concerns of storing data on external clouds. The lack of control over one's own data and the increasing number of cyber threats have necessitated legal requirements for data privacy and security (e.g., GDPR [72] or HIPAA [100]) from third-party cloud providers [89]. When cloud providers fail to meet legal regulations, the authorities fine them in millions to billions of dollars [199, 65]. As a result, cloud providers are being forced to reevaluate their priorities and focus on data security as well while designing cloud based data systems. However, there exists a fundamental trade-off between security and efficiency in data management systems. Although resolving this tension is challenging, it has fostered the growth of a deep field at the intersection of cryptography and database research. In Springer, for example, as of 2021, the number of articles published at this intersection increased by 50.1% since 2015 and by 91.8% since 2000[1]. A database system aiming to provide security guarantees needs to consider three facets of security: *confidentiality,*

[1] We searched for conference papers with keywords database AND security OR privacy OR encryption on https://link.springer.com/search.

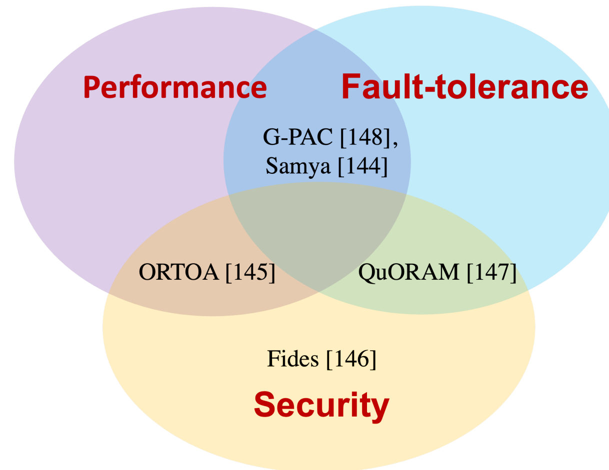

Figure 1.1: An overview of how different data managing systems and protocols presented in the dissertation connect together.

integrity, and availability – generally known as the *CIA* triad. Confidentiality ensures data privacy, integrity guarantees data accuracy and consistency, and availability implies that authorized users have reliable access to data. Providing CIA guarantees while maintaining low data access latency and ensuring data fault tolerance becomes extremely challenging, forming one of the other major difficulties of developing cloud databases.

In summary, designing cloud-based databases for global scale applications entails many challenges including ensuring low data access latency and high performance, data fault tolerance, and data security. Addressing these challenges and improving the performance, fault tolerance, and security of data management systems form the basis of this dissertation.

1.2 Dissertation Overview

This dissertation presents designs and evaluations of data management systems and protocols that strike a balance between performance, fault tolerance, and security. The first question we pose before tackling an issue is: *is the underlying infrastructure that*

hosts the data trusted? Applications that own a large storage and compute fleet can typically trust their data storage infrastructure, in which case enhancing the performance and fault tolerance becomes the main focus when designing database systems. If an application rents storage and compute from an external cloud provider, where the application has limited control on how the external cloud views and manages the data, then the application cannot assume the underlying infrastructure to be fully trusted, in which case we focus developing systems that provide data security. Figure 1.1 presents an overview of the various data managing solutions proposed in the dissertation and how they connect each other. The following sections provide an overview of the systems and protocols presented in the later chapters of the dissertation.

1.2.1 Trusted Infrastructure

Before delving into untrusted settings, this dissertation considers traditional cloud settings, which assume trust. Cloud enterprises typically *replicate* their data to provide fault tolerance, and *shard* (or partition) their data and store the shards on multiple servers to provide scalability. State of the art databases, such as Google's Spanner [45], rely on atomic commit protocols (e.g., Two Phase Commit [87]) to allow scalability and consensus protocols (e.g., Paxos [122]) to achieve replication. Spanner treats atomic commitment and consensus disjointedly, wherein it hides the consensus logic from commitment logic and vice versa. Motivated by the coexistence of sharding and replication in most real-world databases, our work [148] unifies the two seemingly disparate paradigms into a single framework called *Consensus and Commitment* (C&C). The C&C framework is a great pedagogical tool as it can model many established data management protocols, while also providing insight to propose new ones. To highlight its advantages, we propose a novel commit protocol, *G-PAC*, which merges consensus with commitment. As a key feature, unlike many existing protocols that separate failure recovery logic from failure-

free execution logic, G-PAC executes the same set of instructions for both scenarios, easing a developer's job. The unified approach of G-PAC reduces one (out of three) round of cross-datacenter communication compared to Google's Spanner; this allows **G-PAC to perform between 27-88% better than Spanner-like protocols in terms of throughput**.

While G-PAC leveraged the C&C framework to commit distributed transactions for generalized workloads, we developed Samya [144] to extend the C&C framework to manage *high contention, hotspot* data. Samya, a geo-distributed data management system, stores and manages hotspot aggregate data (e.g., number of cloud resources or number of items in stock). State-of-the-art geo-distributed databases such as Google's Spanner [45] employ a centralized approach where a server acting as a *leader* processes all client requests to a specific data item sequentially, which thwarts throughput and leaves the data replicas underutilized. To utilize all replicas or sites that store a data item, Samya splits or partitions the aggregate data, modeled as tokens (e.g., resource tokens), and stores individual token-partitions on different servers. This design choice allows servers in Samya to independently and concurrently serve client requests. A site serves requests locally as long as it has locally available tokens. Samya employs predictive models to predict if a site will exhaust its tokens; if so, the sites execute a novel synchronization protocol, Avantan, extended from the abstractions of the C&C framework, to redistribute the spare tokens in the system. Compared to the centralized solution, **Samya's parallelism reduces the 99th percentile latency by 76% and allows Samya to serve 16x to 18x more requests**.

1.2.2 Untrusted Infrastructure

After working with fully trusted infrastructures, we became intrigued by the question *"What if we host the data on completely untrusted infrastructure?"*. As noted earlier, when storing data on completely untrusted infrastructure, we primarily focus on ensuring data security while striving for the best possible performance. With regard to security, this dissertation proposes three solutions, each addressing one of the challenges of the CIA triad: *confidentiality, integrity,* and *availability.*

Ensuring data integrity: As the first step towards building secure data management systems, we propose *Fides* [146], a transactional DBMS that ensures data integrity, i.e., it ensures the data stored across untrusted servers remains accurate and consistent. Being a transactional system, Fides guarantees *verifiable-ACID*: an external auditor can audit the untrusted storage servers to detect any violations to transactional Atomicity, Consistency, Integrity, or Durability guarantees. As an integral part of Fides, we propose *Trust-Free Commit* (TFCommit), a trust-free commitment protocol that executes transactions across multiple untrusted servers [146]. To our knowledge, ***TFCommit is the first atomic commitment protocol to handle malicious failures without using expensive Byzantine replication***. Byzantine replication protocols tolerate less that $n/3$ malicious failures, where n represents the total number of servers, whereas TFCommit tolerates up to $n-1$ malicious failures. TFCommit combines Two Phase Commit [87] with a collective signature scheme, CoSi [196], to commit transactions across multiple untrusted servers. Similar to blockchain, committing each transaction (or a set of transactions) produces a tamper-resistant log entry, which each server appends to its log. The tamper-resistance stems from collective signature created during commitment that cannot be modified by a single server. An external auditor then audits this log to detect any faulty behavior. TFCommit's failure detection mechanism precisely identifies

the point in the execution history at which a fault occurred as well as the servers that failed. These guarantees provide two fold benefits: (i) An auditor always detects both a malicious fault and the misbehaving database server, and (ii) A benign server can always defend itself against false accusations. Many scenarios can benefit from TFCommit, such as blockchain systems or supply-chain management where the participating entities span different administrative domains that do not trust each other. Compared to executing a transaction in trusted infrastructures, **executing TFCommit has only 1.8x overhead in latency and 2.1x in throughput - an acceptable overhead for many use cases given the additional security guarantees of TFCommit**.

Ensuring data availability: While Fides ensured data integrity, the lack of secure and private datastores that guarantee data fault tolerance inspired us to work on data availability in privacy-preserving systems. Recent attacks revealed the inefficiency of mere data encryption in protecting data privacy [166, 165, 105, 211]; an attacker can uncover non-trivial information either about the data or its users by observing the users' access patterns. Oblivious RAM, or ORAM, a cryptographic technique introduced by Goldreich and Ostrovsky [78], prevents access pattern attacks. Although the database literature consists of many ORAM-based systems, to-date no ORAM-based datastore supports fault tolerance – an important primitive in database systems. To this end, our work *QuORAM* [147], proposes a quorum-based replicated ORAM datastore that tolerates crash failures while preserving obliviousness. QuORAM consists of multiple untrusted and potentially colluding storage servers, each accessed via a separate trusted proxy. QuORAM guarantees linearizable semantics – all operations on a data item appear to be linear – using *a lock-free replication protocol* where a client always reads from a quorum (e.g., majority) of replicas and writes to the same quorum, for both read and write requests. If proxies treat the reads and writes as separate ORAM requests, then they fetch the path twice sequentially from the storage server, which leads to prohibitive latencies.

To avoid double-fetching, the proxies in QuORAM maintain an *incompleteCacheMap* to store request mappings of requests that are read but not yet written back. Our evaluations of QuORAM reveal the advantages of geo-replication in ORAM systems: reduced latency for geo-distributed clients and reduced load on a single proxy. **QuORAM reduces the average data access latency by 61.6% and improves the throughput by 1.4x compared to a non-replicated ORAM system, while providing fault tolerance**.

Ensuring data confidentiality: After developing systems that guarantee data integrity and availability in untrusted settings, we delve into building a low-latency confidentiality-preserving datastore. Given that data encryption alone cannot guarantee complete data privacy, researchers have built solutions such as ORAM [78, 195, 25, 184, 181, 215, 51], Private Information Retrieval (PIR) [42, 120, 75, 32, 25, 134], and frequency smoothing [90] to hide access patterns. In general, access pattern obliviousness in the above schemes consists of two aspects: (i) hiding the exact data item, or rather the exact physical location of the data item accessed by a client; (ii) hiding the *type* of access, i.e., a read vs. a write, requested by a client. To our knowledge, most existing solutions for access pattern obliviousness focus on proposing novel ways to solve aspect (i); while for aspect (ii), the most commonly adapted solution is to always perform a read followed by a write [195, 184, 90, 181], irrespective of the type of request. Always reading followed by writing to hide the type of access incurs *two sequential rounds* of accesses between the clients and the external server resulting in significant overhead; *eliminating this additional overhead is the focus of this work.* To achieve this goal, we propose ORTOA [145], a novel One Round Trip Oblivious Access protocol to access a data item stored on an external untrusted server without revealing the type of access, *in a single round.* This reduction in one round of communication plays a vital role in reducing end-to-end latency, especially in geo-distributed settings. ORTOA's solution is inspired by garbled circuit constructions [217, 132] wherein the external storage server

stores secret labels corresponding to each bit of the plaintext value. To ensure read-write indistinguishability, after each access to a data item, ORTOA updates the labels of that data item. **Our experimental evaluations show that compared to ORTOA a baseline that requires two rounds to hide the type of access incurs 0.76x-1.61x higher latency and 43%-61% lower throughput than ORTOA.**

1.3 Dissertation Organization

This dissertation is organized as follows: In the trusted environments, Chapter 2 presents an abstract unified framework called Consensus & Commitment (C&C) framework that captures both distributed consensus and atomic commit protocols. The chapter also presents G-PAC, an atomic commitment protocol to commit transactions across sharded and partitioned data. Chapter 3 presents Samya, a data system that handles hot spot data and provides high performance for write intensive workloads. It also proposes a novel consensus protocol, Avantan, derived from the abstractions of the C&C framework.

In the untrusted environments, Chapter 4 proposes Fides, a data system that provides data integrity guarantees when the data fully resides on untrusted infrastructure. The chapter also presents the first atomic commitment protocol, TFCommit, that tolerates malicious failures without relying on expensive byzantine replication protocols. Chapter 5 aims to provide data availability of privacy-preserving datastores and presents QuORAM, the first fault-tolerant and quorum-replicated oblivious datastore. Chapter 6 provides data confidentiality guarantees and proposes ORTOA, a one round trip oblivious access protocol that hides the type of access performed by a client. Chapter 7 concludes the systems and protocols presented in this dissertations and Chapter 8 discusses future research directions that can be spawned from the ideas presented in this dissertation.

Chapter 2

Unifying Consensus and Atomic Commitment for Effective Cloud Data Management

2.1 Overview

Data storage in the Cloud needs to be scalable and fault-tolerant. Atomic commitment protocols such as Two Phase Commit (2PC) provide ACID guarantees for transactional access to sharded data and help in achieving scalability. Whereas consensus protocols such as Paxos consistently rep- licate data across different servers and provide fault tolerance. Cloud based datacenters today typically treat the problems of scalability and fault-tolerance disjointedly. In this chapter, we propose a unification of these two different paradigms into one framework called Consensus and Commitment (C&C) framework. The C&C framework can model existing and well known data management protocols as well as propose new ones. We demonstrate the advantages of the C&C framework by developing a new atomic commitment protocol, Paxos Atomic Commit

(PAC), which integrates commitment with recovery in a Paxos-like manner. We also instantiate commit protocols from the C&C framework catered to different Cloud data management techniques. In particular, we propose a novel protocol, Generalized PAC (G-PAC) that integrates atomic commitment and fault tolerance in a cloud paradigm involving both sharding and replication of data. We compare the performance of G-PAC with a Spanner-like protocol, where 2PC is used at the logical data level and Paxos is used for consistent replication of logical data. The experimental results highlight the benefits of combining consensus along with commitment into a single integrated protocol.

2.2 Introduction

The emergent and persistent need for big data management and processing in the cloud has elicited substantial interest in scalable, fault-tolerant data management protocols. Scalability is usually achieved by partitioning or sharding the data across multiple servers. Fault-tolerance is achieved by replicating data on different servers, often, geographically distributed, to ensure recovery from catastrophic failures. Both the management of partitioned data as well as replicated data has been extensively studied for decades in the database and the distributed systems communities. In spite of many proposals to support relaxed notions of consistency across different partitions or replicas, the common wisdom for general purpose applications is to support strong consistency through atomic transactions [133, 87, 128, 189]. The gold standard for executing distributed transactions is two-phase commit (2PC). But 2PC is a blocking protocol even in the presence of mere site failures [87, 189, 128] which led to the three phase commit protocol (3PC) [189] that is non-blocking in the presence of crash failures. But the general version of 3PC is still blocking in the presence of partitioning failures [190].

On the other hand, the distributed systems and cloud computing community have fully embraced Paxos [122, 37] as an efficient asynchronous solution to support full state machine replication across different nodes. Paxos is a leader-based consensus protocol that tolerates crash failures and network partitions as long as majority of the nodes are accessible. As with all consensus protocols, Paxos cannot guarantee termination, in all executions, due to the FLP Impossibility result [62]. The rise of the Cloud paradigm has resulted in the emergence of various protocols that manage partitioned, replicated data sets [45, 143, 116, 88]. Most of these protocols use variants of 2PC for the atomic commitment of transactions and Paxos to support replication of both the data objects as well as the commitment decisions.

Given the need for scalable fault-tolerant data management, and the complex landscape of different protocols, their properties, assumptions as well as their similarities and subtle differences, there is a clear need for a unifying framework that unites and explains this plethora of commitment and consensus protocols. In this chapter, we propose such a unifying framework: the Consensus and Commitment (C&C) framework. Starting with 2PC and Paxos, we propose a standard state machine model to unify the problems of commitment and consensus.

The unifying framework, C&C, makes several contributions. First, it demonstrates in an easy and intuitive manner that 2PC, Paxos and many other large scale data management protocols are in fact different instantiations of the same high level framework. C&C provides a framework to understand these different protocols by highlighting how they differ in the way each implements various phases of the framework. Furthermore, by proving the correctness of the framework, the correctness of the derived protocols is straightforward. The framework is thus both pedagogical as well as instructive.

Second, using the framework, we derive protocols that are either variants of existing protocols or completely novel, with interesting and significant performance characteris-

tics. In particular, we derive several data management protocols for diverse cloud settings. Paxos Atomic Commitment (PAC) is a distributed atomic commitment protocol managing sharded but non-replicated data. PAC, which is a variant of 3PC, integrates crash recovery and normal operations seamlessly in a simple Paxos-like manner. We then derive Replicated-PAC (R-PAC) for fully replicated cloud data management, which is similar to Replicated Commit [143], and demonstrate that protocols like Google's Spanner [45] as well as Gray and Lamport's Paxos Commit [88] are also instantiations of the C&C framework. Finally we propose G-PAC, a novel protocol for sharded replicated architectures, which is similar to other recently proposed hybrid protocols, Janus [161] and TAPIR [227]. G-PAC integrates transaction commitment with the replication of data and reduces transaction commit latencies by avoiding the unnecessary layering of the different functionalities of commitment and consensus.

Many prior works have observed the similarities between the commitment and consensus problems. At the theoretical end, Guerraoui [92], Hadzilacos [95] and Charron-Bost [39] have investigated the relationship between the atomic commitment and consensus problems providing useful insights into the similarities and differences between them. Gray and Lamport [88] observe the similarities and then derive a hybrid protocol. In contrast, the main contribution of this chapter is to *encapsulate* commitment and consensus in a *generic framework*, and then to derive diverse protocols to demonstrate the power of the abstractions presented in the framework. Many of these derived protocols are generalizations of existing protocols, however, some of them are novel in their own right and provide contrasting characteristics that are particularly relevant in modern cloud computing settings.

The chapter is developed in a pedagogical manner. In §2.3, we explain 2PC and Paxos and lay the background for the framework. In §2.4 we propose the C&C unifying framework. This is followed in §2.5 by the derivation of a novel fault-tolerant, non-

Algorithm 1 Given a set of *response values* sent by the cohorts, the coordinator chooses one value for the transaction based on the following conditions.

Possible values are: V = {2PC-yes, 2PC-no}.
Value Discovery: Method $\mathcal{V}$

1: Input: *response values* $\subset$ V *response values from **all** cohorts* 2PC-no $\in$ *response values*
2: *value* $\leftarrow$ *abort*
3: *value* $\leftarrow$ *commit* Timer $\mathcal{T}$ times out
4: * If a cohort crashed *\
5: *value* $\leftarrow$ *abort*

blocking commit protocol, PAC, in a sharding-only environment. In §2.6, we propose R-PAC for atomically committing transaction across fully-replicated data. §2.7 introduces G-PAC for managing data in a hybrid case of sharding and replication. §2.8 provides a safety proof for the C&C framework. In §2.9 we experimentally evaluate the performance of G-PAC and compare it with Spanner-like commit protocol. We discuss the related work in §2.10 and §2.11 concludes the chapter.

2.3 Background

In this section, we provide an overview of 2PC and Paxos as representatives of consensus and atomic commit protocols, respectively. Our goal is to develop a unified framework for a multitude of protocols used in the cloud. In this section, we provide a simple state-machine representation of 2PC and Paxos. These state-machine representations are essential in our framework development later in the chapter.

2.3.1 Two Phase Commit

Two-Phase Commit (2PC) [87, 128] is an atomic commitment protocol. An atomic commitment protocol coordinates between data shards whether to commit or abort a transaction. Commitment protocols typically consist of a *coordinator* and a set of *co-*

horts. The coordinator drives the commitment of a transaction. The cohorts vote on whether they agree to commit or decide to abort. We use the notion of a method $\mathcal{V}$ that takes as input all the votes of individual cohorts and decides on a final value for the transaction. Method $\mathcal{V}$ is represented in Algorithm 1.

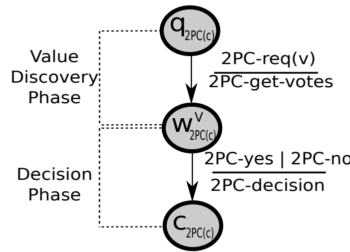

Figure 2.1: State machine representation of a Two Phase Commit coordinator.

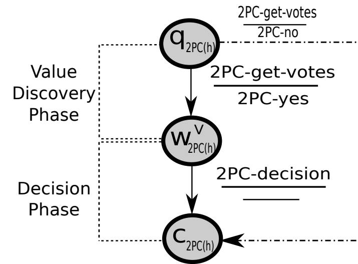

Figure 2.2: State machine representation of a Two Phase Commit cohort.

We now present a state machine that represents 2PC. We show two distinct state machines, one for the coordinator that triggers the commitment protocol, and another for each cohort responding to the requests from the coordinator. In each state machine, a state is represented as a circle. An arrow from a state s^i to s^j with the label $\frac{e_{i,j}}{a_{i,j}}$ denotes that a transition from s^i to s^j is triggered by an event $e_{i,j}$ and causes an action $a_{i,j}$. Typically, events are received messages from clients or other nodes but may also be internal events such as timeouts or user-induced events. Actions are the messages that

are generated in response to the transition, but may also denote state changes such as updating a variable. We use notations q as a starting state, followed by one or more waiting states, denoted by w and the final commit (representing both commit and abort) state, c.

Figures 2.1 and 2.2 show the state machine representation of 2PC. In this work, we represent both transaction commit and transaction abort as one final state, instead of having two different final states for commit and abort decisions as represented in [189]. The coordinator (Figure 2.1) begins at the initial state $q_{2PC(C)}$ (the subscript $2PC(C)$ denotes a 2PC coordinator state). When a client request, 2PC-req[1], to end the transaction arrives, the coordinator sends 2PC-get-votes messages to all cohorts and enters a waiting state, $w^V_{2PC(C)}$. Once all cohorts respond, the responses are sent to method $\mathcal{V}$ represented in Algorithm 1 and a decision is made. The coordinator propagates 2PC-decision messages to all cohorts and reaches the final state $c_{2PC(C)}$. Although 2PC handles asynchronous network, in practice, if the coordinator does not hear back the value from a cohort after a time $\mathcal{T}$, the cohort is considered to be failed and the method $\mathcal{V}$ returns *abort*.

Figure 2.2 shows the state machine representation for a cohort. Initially, the cohort is in the initial state, $q_{2PC(h)}$ (the subscript $2PC(h)$ denotes a 2PC cohort state). If it receives a 2PC-get-votes message from a coordinator, it responds with a yes or no vote. A no vote is a unilateral decision, and therefore the cohort moves to the final state immediately with an abort decision. A yes vote will move the cohort to a waiting state, $w^V_{2PC(h)}$. In both cases, the cohort responds with its vote to the coordinator. While in $w^V_{2PC(h)}$ state, the cohort waits until it receives a 2PC-decision message from the coordinator and it moves to the final decision state of $c_{2PC(h)}$. 2PC can be blocking when there are crashes but we will not discuss it in this work.

[1]The different protocols we discuss use similar terminology for messages with different meanings. To avoid confusion, we will use a prefix to denote the corresponding protocol.

Algorithm 2 Given a set of responses sent by the acceptors, the leader decides whether to transit from one state to another based on the conditions explained below.

Possible responses: {pax-prepared, pax-accept}.

Leader election: Method $\mathcal{M}$

1: $Q_M \leftarrow$ *majority quorum* pax-prepared *messages from* Q_M
2: return *true*
3: return *false*

Replication: Method $\mathcal{R}$

1: $Q_M \leftarrow$ *majority quorum* pax-accept *messages from* Q_M
2: return *true*
3: return *false*

2.3.2 Paxos Consensus Protocol

Paxos [122] is a consensus protocol that is often used to support fault-tolerance through replication. Consensus is the problem of reaching agreement on a single value between a set of nodes. To achieve this, Paxos adopts a leader-based approach. Figures 2.3 and 2.4 present the state machine representation of Paxos: one for the process aspiring to be the leader, called a *proposer*, and another for each process receiving requests from the proposer, called an *acceptor*.

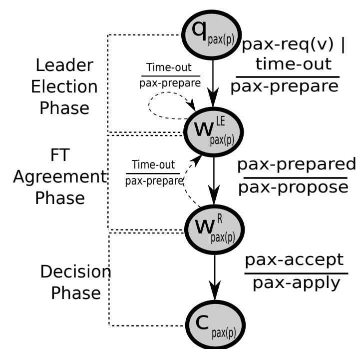

Figure 2.3: State machine representation of a Paxos proposer.

Consider the proposer state machine in Figure 2.3. The proposer is with an initial state $q_{pax(p)}$ (the subscript $pax(p)$ denotes a Paxos proposer state). A user request to

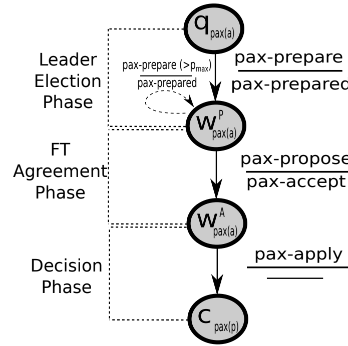

Figure 2.4: State machine representation of a Paxos acceptor.

execute value v, pax-req(v), triggers the Leader Election phase. pax-prepare messages are sent with the proposal number (initially 0) to at least a majority of acceptors. The proposer is then in state $w^{LE}_{pax(p)}$ waiting for pax-prepared messages. The condition to transition from $w^{LE}_{pax(p)}$ state to the next state is given by method $\mathcal{M}$ in Algorithm 2: once a majority of acceptors, denoted by *majority quorum* Q_M, respond with pax-prepared messages, the proposer moves to the Replication phase (state $w^{R}_{pax(p)}$), sending pax-propose messages to at least a majority of acceptors and waiting to receive enough pax-accept messages. To decide the completion of replication phase, the leader uses method $\mathcal{R}$ described in Algorithm 2, which requires a majority of pax-accept messages. The proposer then moves to the final commit state, denoting that the proposed value has been chosen, and pax-apply messages are sent to acceptors, notifying them of the outcome.

An unsuccessful Leader Election or Replication phase may be caused by a majority of acceptors not responding with pax-prepared or pax-accept messages. In these cases, a timeout is triggered (in either state $w^{LE}_{pax(p)}$ or $w^{R}_{pax(p)}$). In such an event, a new unique proposal number is picked that is larger than all proposal numbers received by the proposer. This process continues until the proposer successfully commits the value v.

Now, consider the acceptor state machine in Figure 2.4. Initially, the acceptor is in an initial state $q_{pax(a)}$ (the subscript $pax(a)$ denotes a Paxos acceptor state). The

acceptor maintains the highest promised proposal number, denoted p_{max}. An acceptor may receive a pax-prepare message which triggers responding with a pax-prepared message if the received ballot is greater than p_{max}.. After responding, the acceptor moves to state $w^P_{pax(a)}$ waiting for the next message from the leader. If the acceptor receives a pax-accept message with a proposal number that is greater or equal to p_{max}, then it moves to state $w^A_{pax(a)}$. Finally, the acceptor may receive a pax-apply message with the chosen value.

Note that, for presentation purposes, we omit reactions to events that do not change the state of the process. An example of such reactions is an acceptor responding to a pax-prepare or a pax-propose messages in the commit state. In case of a leader failure while executing Paxos, an acceptor will detect the failure using a timeout. This acceptor now tries to become the new leader, thus following the states shown in Figure 2.3.

2.4 Unifying Consensus and Commitment

In this section, we present a Consensus and Commitment (C&C) framework that unifies Paxos and 2PC. This general framework can then be instantiated to describe a wide range of data management protocols, some of them well known, and others novel in their own right. We start by exploring a high level abstraction of Paxos and 2PC that will aid in developing the framework, and then derive the general C&C framework.

2.4.1 Abstracting Paxos and 2PC

In this section we deconstruct Paxos and 2PC into their basic tasks. Through this deconstruction we identify the tasks performed by both protocols that lead to the construction of the C&C framework.

Both consensus and atomic commit protocols aim at ensuring that *one* outcome is agreed upon in a distributed environment *while tolerating failures*. However, the condi-

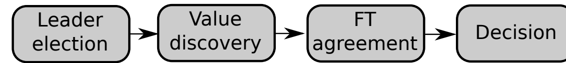

Figure 2.5: A high-level sequence of tasks that is generalized based on Paxos and 2PC.

tions for achieving this agreement is different in the two cases. The basic phase transitions for Paxos are: leader election, followed by fault-tolerant replication of the value and finally the dissemination of the decision made by leader. For 2PC, considering a failure free situation, a predetermined coordinator requests the value to decide on from all cohorts, makes the decision based on the obtained values and disseminates the decision to all cohorts. We combine the above phases of the two protocols and derive a high level overview of the unified framework shown in Figure 2.5. Each of the phases shown in Figure 2.5 is described in detail explaining each phase, its significance and its derivation.

Phases of the C&C Framework

- ***Leader Election***: A normal operation in Paxos encompasses a leader election phase. 2PC, on the other hand, assumes a predesignated leader or a coordinator, and does not include a leader election process as part of normal operation. However, if the coordinator fails while committing a transaction, one way to terminate the transaction is by electing one of the live cohorts as a coordinator which tries to collect the states from other live nodes and attempts to terminate the transaction.

- ***Value Discovery***: Both Paxos and 2PC are used for reaching *agreement* in a distributed system. In Paxos, agreement is on arbitrary values provided by a client, while in 2PC agreement is on the outcome of a transaction. The decision in 2PC relies on the state of the cohorts and hence requires communication with the cohorts. This typically constitutes the first phase of 2PC. Whereas in Paxos, although it is agnostic to the process of deriving the value and the chosen value is independent of the current state of acceptors, it does incorporate value discovery during re-election. The response

to a leader election message inherently includes a previously accepted value and the new leader should choose that value, in order to ensure the safety of a previously decided value.

- ***Fault-Tolerant Agreement***: Fault tolerance is a key feature that has to be ensured by all atomic commitment and consensus protocols. In the most naive approach, 2PC provides fault tolerance by persisting the decision to a log on the hard disk and recovering from the logs after a crash [127]. In Paxos, the role of the replication phase is essentially to guarantee fault tolerance by ensuring that the value chosen by the leader is persistent even in the event of leader failure. As explained in §2.3.2, the value proposed by the leader will be stored in at least a majority of the nodes, thus ensuring fault-tolerance of the agreed upon value.

- ***Decision***: In Paxos, once the leader decides on the proposed value, it propagates the decision *asynchronously* to all the participants who *learn* the decision. Similarly in 2PC, once a decision is made, the coordinator disseminates the decision to all the cohorts. Essentially, in both protocols a value is decided by the leader based on specific conditions and that value (after made fault tolerant) is broadcast to all the remaining nodes in the system.

Given the task abstraction of the C&C framework, we can see that a Paxos instantiation of the framework, in normal operation, will lead to Leader Election, Fault-tolerant (FT) Agreement and Decision phases but will skip the additional Value Discovery phase. On the other hand, a 2PC instantiation of the C&C framework, in normal operation, will become a sequence of Value Discovery and Decision phase, avoiding an explicit Leader Election phase and FT-Agreement phase.

Although we highlighted the high-level similarities in the two protocols, there are subtle differences between the two problems of consensus and commitment. For example: the difference in the number of involved participants in both protocols: Paxos only needs

a majority of nodes to be alive for a decision to be made whereas 2PC needs votes from all the participants to decide on the final value. Such subtleties in different protocols can be captured by specific protocol instantiations of the generic framework.

2.4.2 The C&C Framework

The Consensus and Commitment (C&C) framework aims to provide a general framework that represents both consensus and atomic commitment protocols. In Section 3.1, we started by unifying the two key protocols, Paxos and 2PC, and developed a high level abstraction of the unified framework. Now we expand the precise states in the C&C framework and the transitions across different states. Since our framework takes a centralized approach, each participating node in the system either behaves as a *leader* or a *cohort*. Figures 2.6 and 2.7 show the state machines for a leader and a cohort in the framework. As mentioned earlier, an arrow from states s^i to s^j with the label $\frac{e_{i,j}}{a_{i,j}}$ denotes a transition from s^i to s^j. This transition is triggered by an event $e_{i,j}$ and the transition causes an action $a_{i,j}$. One important point to keep in mind is that each event $e_{i,j}$ and its corresponding action $a_{i,j}$ can have different meaning in different protocol instantiations.

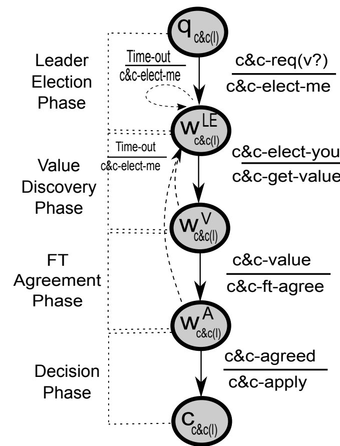

Figure 2.6: State machine representation of a primary node in C&C framework.

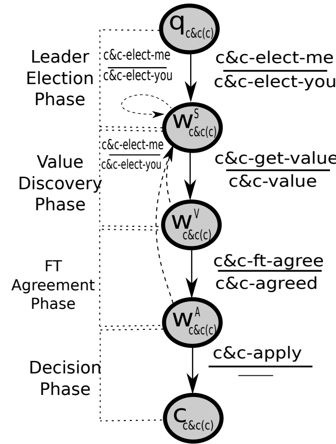

Figure 2.7: State machine representation of a secondary node in C&C framework.

We first define the typical behavior of a leader node in the C&C framework. As shown in Figure 2.6, the leader node starts in the initial state $q_{c\&c(l)}$ (l indicates leader). A client sends c&c-req(v?) request to node $\mathcal{L}$. Depending on the protocol being instantiated, a client request may or may not contain the value v on which to reach agreement. In commitment protocols, the client request will be void of the value. The leader $\mathcal{L}$ increments its ballot number b and starts *Leader Election* by sending c&c-elect-me messages containing b to all the cohorts. The leader waits in the $w^{LE}_{c\&c(l)}$ state for a majority of the cohorts to reply with c&c-elect-you messages. We model the above event as the method $\mathcal{M}$ as explained in Algorithm 2 which will return true when a majority of the cohorts vote for the contending leader. Once $\mathcal{L}$ receives votes from a majority, it starts *Value Discovery*. The c&c-elect-you message can contain previously accepted values by the cohorts, in which case $\mathcal{L}$ chooses one of these values. Otherwise, $\mathcal{L}$ sends c&c-get-value to all participants and transitions to the wait state $w^{V}_{c\&c(l)}$. The leader now waits to receive c&c-value messages from *all* the participants and since value discovery is derived from 2PC, C&C uses the method $\mathcal{V}$, explained in Algorithm 1, to decide on a value based on the c&c-value replies. Method $\mathcal{V}$ can be overridden depending on the requirements of the instantiating protocol (as will be shown in §2.5). The leader makes the chosen

value fault-tolerant by starting *FT-Agreement* and sending out c&c-ft-agree messages to all nodes. $\mathcal{L}$ waits in the $w^{A}_{c\&c(l)}$ state until method $\mathcal{R}$ in Algorithm 2 (with c&c-agreed messages as input) returns *true*. In $\mathcal{R}$ we use majority quorum but an instantiated protocol can use any other quorum as long as the quorum in method $\mathcal{R}$ **intersects** with the quorum used in method $\mathcal{M}$. This follows from the generalizations proposed in [102, 160]. The leader $\mathcal{L}$ finally propagates the decision by sending c&c-apply messages and reaches the final state $c_{c\&c(l)}$.

Now we consider the typical behavior of a cohort in the C&C framework, as shown in Figure 2.7. The cohort $\mathcal{C}$ starts in an initial state $q_{c\&c(c)}$ (c stands for cohort). After receiving a c&c-elect-me message from the leader $\mathcal{L}$, the cohort responds with c&c-elect-you upon verifying if ballot b sent by the leader is the largest ballot seen by $\mathcal{C}$. The c&c-elect-you response can also contain any value previously accepted by $\mathcal{C}$, if any. $\mathcal{C}$ then moves to the $w^{LE}_{c\&c(c)}$ state and waits for the new leader to trigger the next action. Typically, the cohort receives a c&c-get-value request from the leader. Each cohort independently chooses a value and then replies with a c&c-value message to $\mathcal{L}$. In atomic-commitment-like protocols, the value will be either *commit* or *abort* of the ongoing transaction. The cohort then waits in the $w^{V}_{c\&c(c)}$ state to hear back the value chosen by the leader. Upon receiving c&c-ft-agree, the cohort stores the value sent by leader and acknowledges its receipt to the leader by sending c&c-agreed message, and moving to $w^{A}_{c\&c(c)}$ state. Once fault-tolerance is achieved, the leader sends c&c-apply and the cohort applies the decided value and moves to the final state $c_{c\&c(c)}$.

A protocol instantiated from the framework can have either all the state transitions presented in Figures 2.6 and 2.7 or a subset of the states. Specific protocols can also merge two or more states for optimization (PAC undertakes this approach).

Safety in the C&C framework:

Any consensus or commitment protocol derived from the C&C framework should provide safety. **The safety condition states that a value once decided, will never be changed.** A protocol instantiated from the C&C framework will guarantee an overlap in the majority quorum used for Leader Election and the majority quorum used in Fault-Tolerant Agreement. This allows the new leader to learn any previously decided value, if any. We provide a detailed Safety Proof in the Appendix section 2.8 and show that the C&C framework is safe, and thus, we argue that any protocol instantiated from the framework is also safe.

2.5 Sharding-Only in the Cloud

We now consider different data management techniques used in the cloud and derive commitment protocols in each scenario using the unified model. This section deals with the sharding only scenario where the data is sharded and there is no replication. When the data is partitioned, transactions can access data from more than one partition. To provide transactional atomicity, distributed atomic commitment protocols such as 2PC [87] and 3PC [189] are used. Since crash failures are frequent and 2PC can be blocking, 3PC was proposed as a non-blocking commitment protocol under crash failures. 3PC is traditionally presented using two modes of operation: 1) Normal mode (without any failures), 2) Termination mode (to terminate the ongoing transaction when a crash occurs) [22, 189]. 3PC is non-blocking if a majority (or a quorum) of sites are connected. However, Keidar and Dolev [109] show that 3PC may still suffer from blocking after a series of failures even if a quorum is connected. They develop a protocol, E3PC, that guarantees any majority of sites to terminate irrespective of the failure pattern. However, E3PC still requires both normal and failure mode operations. Inspired by the simplicity

of Paxos to integrate the failure-free and crash- recovery cases in a single mode of operation, we use the C&C framework to derive an integrated atomic commitment protocol, PAC, which, similar to E3PC, is guaranteed to terminate as long as a majority of sites are connected irrespective of the failure pattern. The protocol presented in this section and the subsequent ones assume asynchronous networks.

2.5.1 System Model

Transactions accessing various shards consist of read and/ or write operations on the data objects stored in one or more shards. The term *node* abstractly represents either a process or a server responsible for a subset of data. A key point to note here is that the protocol developed in this section (and the subsequent ones) is oblivious to the underlying concurrency control (CC) mechanism. We can use a pessimistic CC such as Two Phase Locking [87] or an optimistic CC technique [119]. The commit protocol is derived such that each data shard has a transaction manager and when the client requests to end a transaction, the underlying transaction manager for each data shard decides to either commit or abort the transaction based on the chosen CC mechanism. Hence, the isolation and serializability guarantees depend on the deployed CC, and is orthogonal to the work presented here, which is focused on the atomic commitment of a *single* transaction, as is traditional with atomic commitment protocols.

2.5.2 Protocol

We now derive the **Paxos Atomic Commitment** (PAC) protocol from the C&C framework. Each committing transaction executes a single instance of PAC, *i.e.*, if there are several concurrent transactions committing at the same time, multiple concurrent PAC instances would be executing *independently*. In PAC, each node involved in a

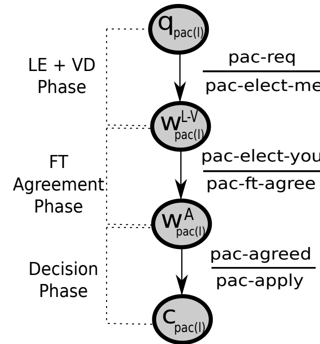

Figure 2.8: State machine representation of a PAC leader.

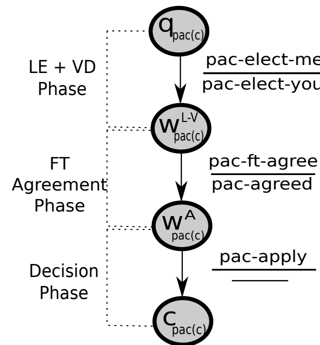

Figure 2.9: State machine representation of a PAC cohort.

transaction T maintains the variables shown in Table 2.1, with their initial values, where p is the process id.

Figures 2.8 and 2.9 show the state transition diagram for both the leader and a cohort in PAC. Abstractly, PAC follows the four phases of the C&C framework: Leader Election, Value Discovery, Fault-tolerant Agreement and the Decision Phases. However, as an optimization, Leader Election and Value Discovery are merged together. Furthermore, in the C&C framework, Leader Election needs response from a *majority* while Value Discovery requires response from *all* the nodes. Hence, the optimized merged phase in PAC needs to receive responses from *all* nodes in the initial round, while any subsequent (after failure) Leader Election may only need to receive response from a *majority*.

BallotNum	initially $< 0, p >$
InitVal	*commit* or *abort*
AcceptNum	initially $< 0, p >$
AcceptVal	initially Null
Decision	initially False

Table 2.1: State variables for each process p in PAC.

Unlike 3PC, in PAC, the external client can send an *end-transaction(T)* request to any shard accessed by T. Let $\mathcal{L}$ be the node that receives the request from the client. $\mathcal{L}$, starting with an initial ballot, tries to become the leader to atomically commit T. Note that in failure-free mode, there are no contending leaders unless the client does not hear from $\mathcal{L}$ for long-enough time and sends the *end_transaction- (T)* request to another shard in T. $\mathcal{L}$ increments its ballot number and sends pac-elect-me messages and moves to the $W^{L-V}_{pac(l)}$ state (l represents leader). A cohort, $\mathcal{C}$, starts in state $q_{pac(c)}$ (c for cohort). After receiving the election message, $\mathcal{C}$ responds with a pac-elect-you message only if $\mathcal{L}$'s ballot number is the highest that $\mathcal{C}$ has received. Based on the CC execution, each node, including $\mathcal{L}$, sets its *InitVal* with a decision *i.e.*, either commit or abort the transaction. The pac-elect-you response contains *InitVal*, any previously accepted value *AcceptVal*, and the corresponding ballot number *AcceptNum*, along with the *Decision* variable (the initial values are defined in Table 2.1).

The transition conditions for $\mathcal{L}$ to move from $W^{L-V}_{pac(l)}$ to $W^{A}_{pac(l)}$ are shown in method $\mathcal{V}$ in Algorithm 3. Once $\mathcal{L}$ receives pac-elect-you messages from a majority of cohorts, it is elected as leader, based on the Leader Election phase of the C&C framework. If none of the responses had *Decision* value true or *AcceptVal* value set, then this corresponds to Value Discovery phase where $\mathcal{L}$ has to wait till it hears pac-elect-you from ALL the cohorts to check if any cohort has decided to abort the transaction. If any *one* cohort replies with *InitVal* as abort, $\mathcal{L}$ chooses abort as *AcceptVal*, and commit otherwise. We will describe crash recovery later. The leader then propagates pac-ft-agree message with the

Algorithm 3 Given the pac-elect-you replies from the cohorts, the leader chooses a value for the transaction based on the conditions presented here.

Leader Election + Value Discovery: Method $\mathcal{V}$

The replies contain variables defined in Table 2.1.

```
1: if received response from a MAJORITY then
2:     if at least ONE response with Decision=True then
3:         AcceptVal ← AcceptVal of that response
4:         Decision ← True
5:     else if at least one response with AcceptVal≠ ⊥ then
6:     \* Decision is True for none in the received responses.*\
7:         AcceptVal ← AcceptVal of the highest AcceptNum
8:     else if received response from ALL cohorts then
9:     \* The normal operation case *\
10:        if all InitVal = commit then
11:            AcceptVal ← commit
12:        else
13:            AcceptVal ← abort
14:    else
15:        AcceptVal ← abort
16: else transaction is blocked
```

chosen *AcceptVal* to all the cohorts and starts the fault-tolerant agreement phase Each cohort upon receiving pac-ft-agree message validates the ballot number and updates the local *AcceptVal* to the value chosen by the leader. It then responds to the leader with pac-ft-agreed message, thus moving to $W^{A}_{pac(c)}$ state.

The leader waits to hear back pac-ft-agreed message from only a majority, as explained in method $\mathcal{R}$ in Algorithm 2 but with pac-ft-agreed messages as input. After hearing back from a majority, the leader sets *Decision* to True, informs the client of the transaction decision and asynchronously sends out pac-apply message with *Decision* as True to all cohorts, eventually reaching the final state $c_{pac(l)}$. A cohort $\mathcal{C}$ can receive pac-apply message when it is in either $W^{L-V}_{pac(c)}$ or $W^{A}_{pac(c)}$ states, upon which it will update its local *Decision* and *AcceptVal* variables and applies the changes made by the transaction to the data shard that $\mathcal{C}$ is responsible for.

In case of leader failure while executing PAC, the recovery is similar to Paxos (Section 2.3.2). A cohort will detect the failure using a timeout and sends out pac-elect-me to all the live nodes and will become the new leader upon receiving a majority of pac-elect-you. The value to be selected by the leader depends on the obtained pac-elect-you messages, as described in method $\mathcal{V}$ in Algorithm 3. If any node, say $\mathcal{N}$ replies with *Decision* as True, this implies that the previous leader had made the decision and propagated it to at least one node before crashing; so the new leader will choose the value sent by $\mathcal{N}$. If none of the replies has *Decision* as True but at least one of them has *AcceptVal* as commit (or abort), this implies that the previous leader obtained replies from all and had chosen commit (or abort) and sent out pac-ft-agree messages. Hence, the new leader will choose commit (or abort). In all the other cases, abort is chosen. If the new leader does not get a majority of pac-elect-you, then the protocol is blocked. The subsequent phases of replication and decision follow the states shown in Figures 2.8 and 2.9.

2.6 Replication-Only in the Cloud

In this section, we explore a data management technique that deals with fully replicated data *i.e.*, the data is fully replicated across, potentially different, data-centers. Using the unified C&C framework, we derive a commit protocol, similar to PAC, called Replicated-PAC (R-PAC).

2.6.1 System Model

In a traditional state machine replication (SMR) system, the client proposes a value and all replicas try to reach agreement on that value. In a fully replicated data management system, each node in the system maintains an identical copy of the data. Clients perform transactional accesses on the data. Here, the abstraction of a node can represent

a datacenter or a single server; but all entities denoted as nodes handle identical data. At transaction commitment, each node independently decides whether to commit or to abort the transaction. The transactions can span multiple data objects in each node and any updates on the data by a transaction will be executed atomically. Since the data is fully replicated, R-PAC is comparable to the Replicated Commit protocol [143].

Every node runs a concurrency control (CC) protocol and provides a commitment value for a transaction. If a node is an abstraction for a single machine, we can have a CC, such as 2PL, that decides if a transaction can be atomically committed or if it has to be aborted. Whereas, if a node represents a datacenter and if data is partitioned across different machines within the datacenter, the commitment value per node can be obtained in two ways. In the first approach with shared-nothing architecture, each data center has a transaction manager which internally performs a distributed atomic commitment such as 2PC or PAC (§2.5) and provide a single value for that node (datacenter). In the second approach with a shared-storage architecture, different machines can access the same storage driver and detect any concurrency violations [22]. In either architecture, each node provides a *single* value per transaction. For simplicity, we do not delve deeper into the ways of providing CC; rather we work with the abstraction that when the leader asks for a commitment value of a transaction, each *node* provides one value. The protocol presented below ensures that all the nodes either atomically commit or abort the transaction, thus maintaining a consistent view of data at all nodes.

2.6.2 Protocol

The commit protocol, Replicated-PAC (R-PAC), is similar to PAC except for one change: the Value Discovery method $\mathcal{V}$. In PAC (§2.5), during value discovery, the leader waits to receive pac-elect-you message from *all* cohorts. When the data is partitioned, a

commit decision cannot be made until all cohorts vote because each cohort is responsible for a disjoint set of data. In the replication-only case, since all nodes maintain identical data, the leader need to only wait for replies from a *majority* of replicas that have the **same** *InitVal*. Hence, the method $\mathcal{V}$ presented in Algorithm 3 differs for R-PAC only at line 8, namely waiting for pac-elect-you messages from a *majority* rather than *all* cohorts. Since R-PAC only requires a majority of replicas to respond, it is similar to the original majority consensus protocol proposed by Thomas [201]. The rest of the replication phase, decision phase and in case of a crash, the recovery mechanism, are identical to PAC.

Depending on the CC mechanism adopted, and due to the asynchrony of the system, different replicas can end up choosing different commitment values. A key observation here is that if a majority of the replicas choose to commit and one or more replicas in the minority choose to abort the transaction, the leader forces *ALL* replicas to commit the transaction. This does not violate the isolation and serializabilty guarantees of the CC protocol, as updates will not be reflected on the data of a replica R until R receives a pac-apply message. This ensures that all replicas have consistent views of the data at the end of each transaction.

2.7 Sharding + Replication in the Cloud

In this section, we present a general hybrid of the two previously presented data management schemes *i.e.*, data is both partitioned across different shards and each shard is replicated across different nodes. Transactions can span multiple shards, and any update of a shard will cause an update of its replicas. When the data is both sharded and replicated, solutions like Spanner and MDCC [45, 116] use a hierarchical approach by horizontally sharding the data and vertically replicating each shard onto replicas. The replication is managed by servers called *shard leaders*. The other category of solutions,

such as Janus [161] and TAPIR [227] deconstruct the hierarchy of shards and replicas and atomically access data from all the involved nodes. Hence, we categorize the hybrid case of sharded and replicated data into two different classes based on the type of replication: 1) Using standard State Machine Replication (SMR) wherein the coordinator communicates only with the leaders of each shard, 2) Using PAC-like protocol wherein the coordinator of a transaction communicates with all the involved nodes.

2.7.1 Replication using standard SMR: Layered architecture

A popular approach for providing non-blocking behavior to a commitment protocol such as 2PC is to replicate each state of a participating node (coordinator or cohort). SMR ensures fault-tolerance by replicating each 2PC state of a shard to a set of replica nodes. This has been adopted by many systems including Spanner [45] and others [76, 143, 116]. We will refer to this approach as 2PC/SMR. 2PC/SMR shares the objective of 3PC which is to make 2PC fault-tolerant. **While SMR uses replicas to provide fault tolerance, 3PC uses participants to provide persistence of decision**. Therefore, 2PC/SMR can be considered as an alternative to commitment protocols that provides fault tolerance using as additional phase such as 3PC or PAC. 2PC/SMR follows the abstraction defined by the C&C framework. In particular, 2PC provides the Value Discovery phase of C&C and any SMR protocol, such as Paxos, provides the Fault-Tolerant Agreement phase of the framework. The Decision phase of C&C is propagated hierarchically by the coordinator to SMR leaders and then the leaders propagate the decision on to the replicas.

The system model of 2PC/SMR consists of a coordinator and a set of cohorts, each responsible for a data shard. Along with that, 2PC/SMR also introduces *SMR replicas.* SMR replicas are not involved in the 2PC protocol. Rather, they serve as backups for

the coordinator and cohorts and are used to implement the FT-Agreement phase of C&C. The coordinator and cohorts each have a set of—potentially overlapping—SMR replicas, the idea originally proposed by Gray and Lamport in [88]. If a shard holder (coordinator or cohort) fails, the associated SMR replicas recover the state of the failed shard. This changes the state transitions of 2PC. At a high level, every state change in 2PC is transformed into two state changes: one to replicate the state to the associated SMR replicas and another to make the transition to the next 2PC state. For example, before a cohort in 2PC responds to 2PC-get-votes, it replicates its value onto a majority of SMR replicas. Similarly, the coordinator, after making a decision, first replicates it on a majority of SMR replicas before responding to the client or informing other cohorts.

The advantage of using 2PC/SMR in building a system is that if the underlying SMR replication is in place, it is easy to leverage the SMR system to derive a fault-tolerant commitment protocol using 2PC. Google's Spanner is one such example. We discuss the trade-offs in terms of number of communication rounds for a layered solution vs. flattened solution in the evaluation §2.9.

2.7.2 Replication using Generalized PAC: Flattened architecture

In this section, we propose a novel integrated approach for SMR in environments with both sharding and replication. Our approach is a generalized version of PAC, hence is named Generalized PAC or G-PAC. The main motivation driving our proposal is to reduce the number of wide-area communication messages. One such opportunity stems from the hierarchy that is imposed in traditional SMR systems—such as Spanner. The hierarchy of traditional SMR systems incur wide-area communication unnecessarily. This is because the 2PC (Atomic Commitment) layer and Paxos (consensus/replication) layers

are operating independently from each other. Specifically, a wide-area communication message that is sent as part of the 2PC protocol can be used for Paxos (*e.g.*, leader election). We investigate this opportunity and propose G-PAC to find optimization opportunities between the Atomic Commit and Consensus layers.

Algorithm 4 For G-PAC, given the plac-elect-you replies from the participating servers, the leader chooses a value for the transaction based on the conditions presented here.

Leader Election + Value Discovery: Method $\mathcal{V}$
The replies contain variables defined in Table 2.1.

1: **if** received response from a SUPER-MAJORITY **then**
2:     **if** at least ONE shard response has *Decision*=True **then**
3:         $AcceptVal \leftarrow AcceptVal$ of that response
4:         $Decision \leftarrow$ True
5:     **else if** a majority of replicas of at least one shard respond and at least one of them has $AcceptVal \neq \bot$ **then**
    * *Decision* is True for none in the SUPER-MAJORITY.*\
6:         AcceptVal $\leftarrow$ *AcceptVal* of the highest *AcceptNum* across all the received responses
7:     **else if** received response from SUPER-SET **then**
8:    * The normal operation case *\
9:         **if** all $InitVal = commit$
10:             $AcceptVal \leftarrow commit$
11:         **else**
12:             $AcceptVal \leftarrow abort$
13:     **else**
14:         $AcceptVal \leftarrow abort$
15: **else** transaction is blocked

The G-PAC protocol consists of three phases: an integrated Leader Election and Value Discovery phase, a Fault-Tolerant Agreement phase, followed by the Decision phase. If a transaction, T, accesses n shards and each shard is replicated in r servers, there are a total of $n * r$ servers that are involved in the transaction T. This set of servers will be referred as *participating servers*. The client chooses one of the participating servers, $\mathcal{L}$, and sends an *end_transaction(T)* request. $\mathcal{L}$ then tries to become the leader or the coordinator for transaction T. The coordinator, $\mathcal{L}$, and the cohorts follow the same

state transition as shown in Figures 2.8 and 2.9 except that the contending leader sends plac-elect-me message to *all* the participating servers. The overridden Value Discovery method $\mathcal{V}$ is similar to the one presented Algorithm 3. The flattening of the architecture for sharding and replication changes the notion of *all* and *majority* cohorts that is referred in Algorithm 3 to:

- **super-set**: Given n shards, each with r replicas, super -set is a majority of replicas for each of the n shards. The value for each shard is the one chosen by a majority of replicas of that shard *i.e.*, $(\frac{r}{2}+1)$ replicas. If any shard (represented by a majority of its replicas) chooses to abort, the coordinator sets $AcceptVal$ to abort.
- **super-majority**: a majority of replicas for a *majority* of shards involved in transaction T *i.e.*, $(\frac{n}{2}+1)$ shards and for each shard, a value is obtained when a majority of its replicas respond *i.e.*, $(\frac{r}{2}+1)$ replicas for each shard.

Method $\mathcal{V}$ with the newly defined notions of *super-set* and *super-majority*, which decides the value for the transaction, in described in Algorithm 4. We reuse the definition of replication method $\mathcal{R}$ described in Algorithm 2, where majority is replaced by *super-majority*. Note that during the integrated Leader Election and Value Discovery phase, if a node receives response from a *super-majority*, it could be elected as leader, however to proceed with Value Discovery, it needs to wait for a *super-set*, which is a more stringent condition due to the atomic commitment requirement.

2.8 Safety in the C&C framework

Proof: In this section we discuss the safety guarantees that any consensus or commit protocol derived from the C&C framework will provide. **The safety condition sta-tes that a value once decided, will never be changed.** Although majority

quorums are used to explain the state transitions of the C&C framework, for the safety proof we do not assume any specific form of quorum.

Let $\mathcal{Q}_L$ be the set of all possible leader election quorums used in the Leader Election phase and $\mathcal{Q}_R$ be the set of all possible replication quorums used in the Fault-Tolerant Agreement phase of the C&C framework. Any protocol instantiated from the C&C framework should satisfy the following *intersection condition*:

$$\forall Q_L \in \mathcal{Q}_L, \forall Q_R \in \mathcal{Q}_R : Q_L \cap Q_R \neq \emptyset \tag{2.1}$$

The safety condition states that: If a value v is decided for ballot number b, and if a value v' is decided for another ballot number b', then v=v'.

Let $\mathcal{L}$ be the leader elected with ballot number b and v be the value chosen by method $\mathcal{V}$ based on the c&c-value responses. For the chosen value v to be decided, v must be fault-tolerantly replicated on a quorum $Q_R \in \mathcal{Q}_R$.

Now consider another node $\mathcal{L}'$ decides to become leader. $\mathcal{L}'$ sends out c&c-elect-me message with ballot b' to all the other nodes. $\mathcal{L}'$ becomes a leader if it receives c&c-elect-you messages from a quorum $Q_L \in \mathcal{Q}_L$. Based on condition 2.1, $Q_L \cap Q_R$ is non-empty *i.e.*, there is at least one node $\mathcal{A}$ such that $\mathcal{A} \in Q_L$ and $\mathcal{A} \in Q_R$. There can be two possibilities for ballot b'.

- **$b' < b$:** In this case, $\mathcal{L}'$ will not be able to get c&c-elect-you replies from a quorum Q_L as there is at least one node $\mathcal{A}$ that a ballot $b > b'$ and hence will reject $\mathcal{L}'$'s message.
- **$b' > b$:** In this case, as a response to $\mathcal{L}'$'s c&c-elect-me message, $\mathcal{A}$ sends the previously accepted value v to the new leader. $\mathcal{L}'$ then updates the value to propose from v' to v.

Hence, we show that the C&C framework is safe and any protocol instantiated from the framework will be safe as long as condition 2.1 is satisfied. ■

2.9 Evaluation

In our evaluations we compare the performance of G-PAC and 2PC/SMR and discuss the trade-offs in wide-area communication delays between the two approaches. The performance of 2PC/SMR varies widely based on the placement of the SMR leaders. When the SMR leaders are dispersed across different datacenters, the commitment of a transaction needs 4 inter-datacenter round-trip time (RTT) delays: two rounds for the two phases of 2PC and one round each for replicating each of those phases using multi-Paxos [37]. As an optimization, placing all the leaders in a single datacenter will reduce the inter-datacenter communication to 3 RTTs. G-PAC, on the other hand, always only needs 3 RTTs (one for each phase of G-PAC) to complete a transaction commitment.

The practical implications of using G-PAC is that in G-PAC, the leader should know not only the involved shards in a transaction, but also about their replicas. This requires the replicas to have additional information which either has to be stored as meta-data for each transaction or can be stored in a configuration file. If a replica is added/removed or if new shards are added, this will require a configuration change. Propagating this change can be challenging, potentially leading to additional overhead. Although this is practically challenging, many existing practical deployments deal with configuration changes using asynchronous but consistent roll-out based methods such as Raft [172] or Lamport proposals [125, 124]. Any such method can be adapted in the case of G-PAC.

Our experiments evaluate the performance of G-PAC with respect to two versions of 2PC/SMR: the most optimal one (collocated SMR leaders) and the worst-case (geographically distributed SMR leaders across different datacenters). Hence, on average, the performance of 2PC/SMR would lie in between the two cases. We compare the behavior of all the protocols by increasing the concurrent clients (or threads), each of which generates transactions sequentially. We leveraged Amazon EC2 machines from 5 different

	V	I	SP	T
C	60.3	150	201	111
V	-	74.4	139	171
I	-	-	183	223
SP	-	-	-	269

Table 2.2: RTT latencies across different datacenters in ms.

datacenters for the experiments. The datacenters used were N.California (C), N.Virginia (V), Ireland (I), Sao Paolo (SP) and Tokyo (T). In what follows we use the capitalized first initial of each datacenter as its label. Cross datacenter round trip latencies are shown in Table 2.2. In these experiments, we used compute optimized EC2 c4.large machines with 2 vCPUs and 3.75 GiB RAM.

Although G-PAC can be built upon any concurrency control, for equivalent comparison, both G-PAC and 2PC/SMR implementations used **Two Phase Locking** (2PL) as a concurrency control technique. In 2PC/SMR, only the shard leaders maintain the lock table; whereas in G-PAC, all participating servers maintain their own lock tables. Both protocols execute the decision phase asynchronously.

2.9.1 Micro Benchmark

As the first step in our evaluation, we performed the experiments using a transactional YCSB-like [44] micro- benchmark that generated read-write transactions. Every operation within a transaction accessed different keys, thus, generating multi-record transactional workloads. Each shard consisted of 10000 data items. Every run generated 2500 transactions; each plotted data point is an average of 3 runs. To imitate real global data access patterns, the transactions perform a skewed data access *i.e*, 90% of the transactions access 10% of the data objects, while the remaining transactions access the other 90% of the data items.

Varying number of shards

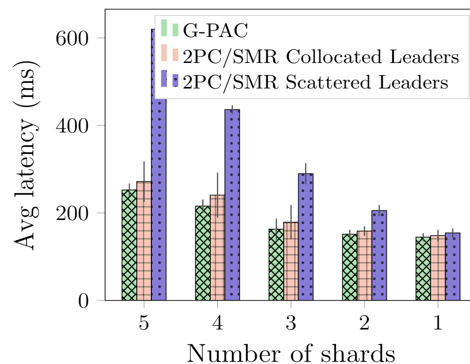

Figure 2.10: Commit latency vs. number of shards.

Commit latency, as measured by the client, is the time taken to commit a transaction once the client sends an *end_ transaction* request. As the first set of experiments, we assessed the average commit latency of transactions when the transactions spanned increasing number of shards. In this experiment, both the coordinator and the client are located on datacenter C. We measure the average latencies for the three protocols by increasing the number of shards accessed by transactions from 1 to 5. And the data was **partially replicated** *i.e.,* each shard was replicated in only 3 datacenters. The results are depicted in Figure 2.10.

When the data objects accessed by the transactions are from all 5 shards (at datacenters C, V, I, SP and T), 2PC/ SMR with leaders scattered on 5 different datacenters, has the highest commit latency, as the coordinator is required to communicate with geo-distributed shard leaders. For 2PC/SMR with scattered leaders, the average commit latency decreases with the reduction in the number of involved shards. This is because with each reduction in number of shards, we removed the farthest datacenter from the experiment. Whereas, the average commit latency for G-PAC does not increase substantially with increasing shards as it communicates only with the closest replicas of each

shard, before responding to the client. When the clients access data from a single shard, the average latencies for all three protocols converge almost to the same value. This is because with a single shard, all three protocols need to communicate with only a majority of the 3 replicas for the shard.

This experiment not only shows that G-PAC has highly stabilized performance when the number of involved shards increase, but it also highlights the fact that, with 2PC/SMR with collocated leaders, at least one datacenter must be over-loaded with all shards in order to obtain optimal results. This is in contrast to G-PAC which equally distributes the load on all datacenters, while preserving optimal latency.

From the results shown in Figure 2.10, we choose 3 shards to run each of the experiments that follow, as it is representative of the trade-offs offered by the three protocols.

2.9.2 TPC-C Benchmark

As a next step, we evaluate G-PAC using TPC-C benchmark, which is a standard for benchmarking OLTP systems. TPC-C includes 9 different tables and 5 types of transactions that access the data stored in the tables. There are 3 read-write transactions and 2 read-only transactions in the benchmark. In our evaluation, we used 3 warehouses, each with 10 districts, each of which in-turn maintained 3000 customer details (as dictated by the spec). One change we adapted was to allocate disjoint sets of items to different warehouses, as the overall goal is to evaluate the protocols for distributed, multi-record transactions. Hence, each warehouse consisted of 33,333 items and the New Order transaction (which consisted of 70% of the requests) accessed items from all 3 warehouses. Each run consisted of 2500 transactions and every data point presented in all the experiments is an average of three runs.

We measured various aspects of the performance of G-PAC and contrasted it with the two variations of 2PC/SMR. We used AWS on 3 different datacenters for the following experiments: C, V and I. In 2PC/SMR with dispersed leaders, the three shard leaders are placed on 3 different datacenters. And for 2PC/SMR with collocated leaders, all shard leaders were placed in datacenter C. Although not required, for ease of evaluation, each shard was replicated across all three datacenters.

Commit Latency

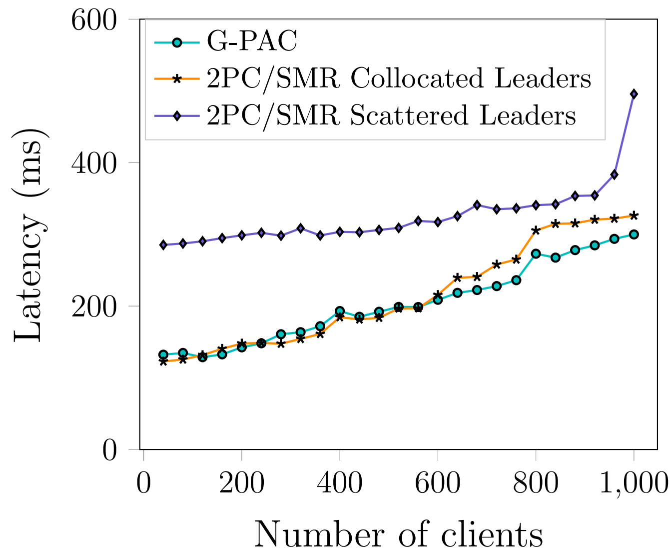

Figure 2.11: Commit latency.

In this experiment, we measure the commit latencies for G-PAC and the two versions of 2PC/SMR while increasing the number of concurrent clients from 20 to 1000. The results are shown in Figure 2.11. Both G-PAC and the optimized 2PC/SMR respond to the client after two RTTs (as the decision is sent asynchronously to replicas), and hence, both protocols start off with almost the same latency values for lower concurrency levels. But with high concurrency, 2PC/SMR has higher latency as all commitments need to go through the leaders, which can become a bottleneck for highly concurrent workloads.

2PC/SMR with dispersed leaders is the least efficient with the highest latency. This is because the coordinator of each transaction needs at least one round of communication with all geo-distributed leaders for the first phase of 2PC (2PC-get-value and 2PC-value). Hence, we observed that G-PAC, when compared to the most and the least performance efficient versions of 2PC/SMR, provides the lowest commit latency of the three.

Number of Commits

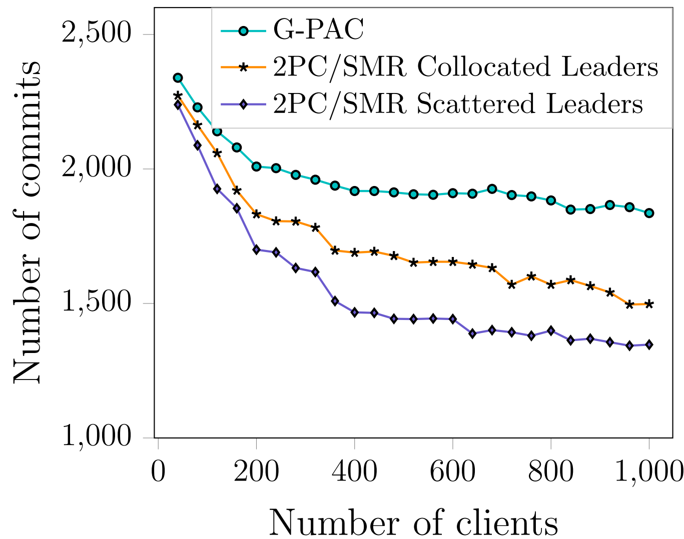

Figure 2.12: Number of commits.

In this set of experiments, we measured the total number of committed transactions out of 2500 transactions by all three protocols while increasing the number of concurrent clients from 20 to 1000. The results are shown in Figure 2.12. From the graph, we observe that G-PAC commits, on an average, 15.58% more transactions than 2PC/SMR with collocated leaders and and 32.57% more than 2PC/SMR with scattered leaders.

Both G-PAC and 2PC/SMR implemented 2-Phase Locking for concurrency control among contending transactions. The locks acquired by one transaction are released after the decision is propagated by the coordinator of the transaction. The leader based layered

architecture of 2PC/ SMR, and its disjoint phases of commitment and consensus, takes an additional round-trip communication before it can release the locks, as compared to G-PAC. And in 2PC/SMR, since lock tables are maintained only at the leaders, at higher contention, more transactions end up aborted. Hence, this experiment shows that flattening the architecture by one level and off-loading the concurrency overhead to all the replicas, G-PAC can release the locks obtained by a transaction earlier than 2PC/SMR, and thus can commit more transactions than 2PC/SMR.

Throughput

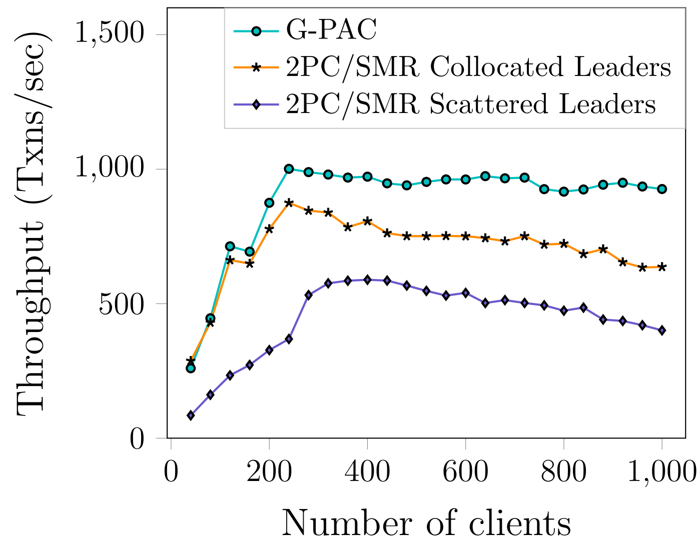

Figure 2.13: Throughput.

We next show the throughput measurements for G-PAC and 2PC/SMR. Throughput is measured as number of successful transaction commits per second and hence, the high contention of data access affects the throughput of the system. Figure 2.13 shows the performance as measured by transactions executed per second with increasing number of concurrent clients. G-PAC provides 27.37% more throughput on an average than 2PC/SMR with collocated leaders and 88.91% higher throughput than 2PC/SMR

with scattered leaders, thus indicating that G-PAC has significantly higher throughput than 2PC/SMR. Although Figure 2.11 showed similar commit latencies for G-PAC and optimized 2PC/SMR, the throughput difference between the two is large. This behavior is due to lower number of successful transaction commits for 2PC/SMR, as seen in Figure 2.12. The scattered leader approach for 2PC/SMR provides low throughput due to larger commit delays. The lower latencies along with greater number of successful transactions boosts the throughput of G-PAC as compared to 2PC/SMR.

Latency of each phase in G-PAC

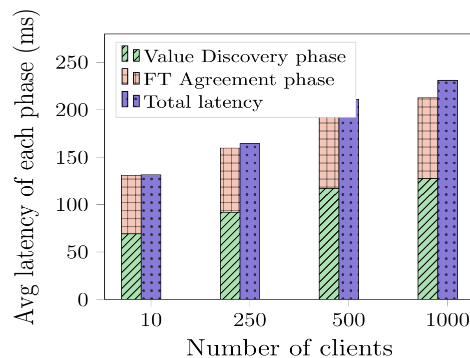

Figure 2.14: Latency of each phase in G-PAC

G-PAC consists of 3 phases: Value Discovery, Fault Tolerant Agreement (essentially replicating the decision) and the Decision phase. In our implementation, the decision is first disseminated to the client, and then asynchronously sent to the cohorts. In this experiment, we show the breakdown of the commit latency and analyze the time spent during each phase of G-PAC. Figure 2.14 shows the average latency spent during each phase, as well as the overall commit latency, with low to high concurrency. The results indicate that the majority of the time is spent on Value Discovery phase (which requires response from super-set of replicas as well as involves acquiring locks using 2PL) and the

FT-Agreement time is quite consistent throughout the experiment (which needs responses only from super-majority and does not involve locking). The increased concurrency adds additional delays to the overall commit latency.

2.9.3 Availability Analysis

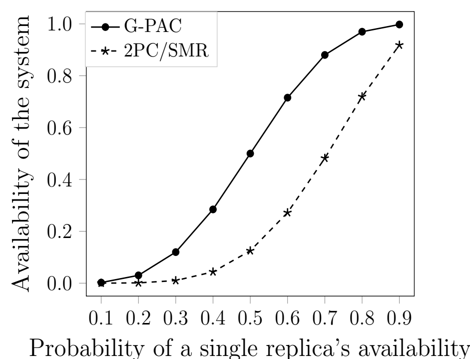

Figure 2.15: Availability Analysis

In this section, we perform a simple availability analysis to better understand how the availability of G-PAC and 2PC/SMR vary with the availability of individual replicas. Let p be the probability with which an individual replica is available. Consider a system involving three shards, where each shard is replicated three ways (2*f+1 with $f = 1$). For the Value Discovery phase, both protocols require a majority of replicas from *all* the shards, hence there is no difference in their availability. But to replicate the decision, G-PAC needs only a super-majority (majority of majority) while 2PC/SMR requires a majority from *all* the shards.

We mathematically represent the availability of both the protocols. First, the availability of each shard (each of which is replicated 3 ways) is computed as shown in Equation 2.2: either all 3 replicas of the shard are available or a majority of the replicas are available. Second, based on the probability, p_{shard}, of each shard being available, we de-

rive the availability of the two protocols. In G-PAC, the decision needs to be replicated in a majority of shards. Provided there are 3 shards, the availability of G-PAC is given by Equation 2.3: either all shards are alive or a majority of them are alive. Similarly, the availability of 2PC/SMR is given by Equation 2.4 which indicates that *all* shards need to be alive to replicate the decision in 2PC/SMR.

$$p_{shard} = 3C_3 * p^3 + 3C_2 p^2 * (1 - p) \tag{2.2}$$

$$A_{G_PAC} = 3C_3 * p_{shard}^3 + 3C_2 p_{shard}^2 * (1 - p_{shard}) \tag{2.3}$$

$$A_{2PC/SMR} = 3C_3 * p_{shard}^3 \tag{2.4}$$

Figure 2.15, shows the availability of G-PAC and 2PC/SMR with increasing probability of an individual site being available. The analysis indicates that G-PAC has higher tolerance to failures than 2PC/SMR. In particular, G-PAC achieves four nines of availability (i.e., 99.99%) when each replica is available with probability $p = 0.96$, where as to achieve the same, 2PC/SMR requires each replica to be available with probability $p = 0.997$.

2.10 Related Work

Distributed transaction management and replication protocols have been studied extensively [22, 213, 122, 158]. These works involve many different aspects, including but not restricted to concurrency control, recovery, quorums, commitment etc. Our focus in this chapter has been on the particular aspect of transaction commitment and how this

relates to the consensus problem. The other aspects are often orthogonal to the actual mechanisms of how the transaction commits. Furthermore, the renewed interest in this field has been driven by the recent adoption of replication and transaction management in large scale cloud enterprises. Hence, in this section we focus on commitment and consensus protocols that are specifically appropriate for Cloud settings that require both sharding and replication, and contrast them with some of the proposed protocols derived from C&C.

We start-off by discussing one of the early and landmark solutions for integrating Paxos and Atomic Commitment, namely, Paxos Commit, proposed by Gray and Lamport [88]. Abstractly, Paxos Commit uses Paxos to fault-tolerantly store the commitment state of 2PC on multiple replicas. Paxos Commit optimizes on the number of message exchanges by collocating multiple replicas on the same node. This is quite similar to the 2PC/SMR protocol of §2.7.1.

Google Spanner [45] adapts an approach similar to Paxos Commit to perform transactional commitment but unlike Paxos Commit, Spanner replicates on geo-distributed ser-vers. 2PC/SMR, developed in §2.7.1 is a high level abstraction of Spanner. Replicated Commit by Mahmoud et al. [143] is a commit protocol that is comparable to Spanner, but unlike Spanner, it assumes full replication of data. Replicated Commit can also be viewed as an instance of the C&C framework, as it can be materialized from the R-PAC protocol (§2.6). MDCC by Kraska et al. [116] is another commit protocol for geo-replicated transactions. MDCC guarantees commitment in one cross datacenter round trip for a collision free transaction. However, in the presence of collision, MDCC requires two message rounds for commitment. Furthermore, MDCC restricts the ability of a client to abort a transaction once the end transaction request has been sent.

More recently, there have been some attempts to consolidate the commitment and consensus paradigms. One such work is TAPIR by Zhang et al. [227]. TAPIR identi-

fies the expensive redundancy caused due to consistency guarantees provided by both the commitment and the replication layer. TAPIR can be specified by the abstractions defined in the C&C framework. In a failure-free and contention-free case, TAPIR uses a *fast-path* to commit transactions, where the coordinator communicates with $\frac{3}{2}f + 1$ replicas of each shard to get the value of the transaction. This follows the Value Discovery phase of the C&C framework. G-PAC contrasts with TAPIR mainly in failure recovery during a coordinator crash. TAPIR executes an explicit *cooperative termination protocol* to terminate a transaction after a crash whereas G-PAC has the recovery tightly integrated in its normal execution. There are other subtle differences between G-PAC and TAPIR: TAPIR does not allow aborting a transaction by the coordinator once commitment is triggered. And although *fast paths* provide an optimization over G-PAC, in a contentious workload, TAPIR's *slow paths* make the complexity of both protocols comparable. Finally, G-PAC provides flexibility in choosing any quorum definition across different phases, unlike the *fast-path* quorum $(3/2f{+}1)$ in TAPIR.

Janus by Mu et al. [161] in another work attempting towards combining commitment and consensus in a distributed setting. In Janus, the commitment of a conflict free transaction needs one round of cross-datacenter message exchange to commit, and with conflicts, it needs two rounds. Although Janus provides low round-trip delays in conflict-free scenarios, the protocol is designed for *stored procedures*. The protocol also requires explicit *a priori* knowledge of write sets in order to construct conflict graphs, which are used for consistently ordering transactions. In comparison, G-PAC is more general, as it does not make any of the assumptions required by Janus.

Another on-going line of work is on deterministic data- bases, where the distributed execution of transactions are planned a priori to increase the scalability of the system. Calvin [202] is one such example. However, this planning requires declaring the read and write sets before the processing of each transaction. This limits the applicability

of deterministic approaches, whereas G-PAC is proposed as a more generalized atomic commit protocol that can be built on top of any transactional concurrency mechanism.

2.11 Conclusion

A plethora of consensus, replication and commitment protocols developed in the past years poses a need to study their similarities and differences and to unify them into a generic framework. In this chapter, using Paxos and 2PC as the underlying consensus and commit protocols, we construct a Consensus and Commitment (C&C) unification framework. The C&C framework is developed to model many existing data management solutions for the Cloud and also aid in developing new ones. This abstraction pedagogically helps explain and appreciate the subtle similarities and differences between different protocols. We demonstrate the benefits of the C&C framework by instantiating a number of novel or existing protocols and argue that the seemingly simple abstractions presented in the framework capture the essential requirements of many important distributed protocols. The chapter also presents an instantiation of a novel distributed atomic commit protocol, Generalized-Paxos Atomic Commit (G-PAC), catering to sharded and replicated data. We claim that separating fault-tolerant replication from the transaction commitment mechanism can be expensive and provide an integrated replication mechanism in G-PAC. We conclude the chapter by evaluating the performance of G-PAC with a Spanner-like solution and highlight the performance gains in consolidating consensus with commitment.

Chapter 3

Samya: Geo-Distributed Data System for High Contention Data Aggregates

3.1 Overview

Geo-distributed databases are the state of the art to manage data in the cloud. But maintaining hot records in geo-distributed databases such as Google's Spanner can be expensive, as it synchronizes each update across a majority of replicas. Frequent synchronization poses an obstacle to achieve high throughput for contentious update-heavy workloads. While such synchronizations are inevitable for complex data types, simple data types such as aggregate data can benefit from reduced synchronizations. In this chapter, we propose an alternate data management system, *Samya*, to store and maintain aggregate data. It is presented as a system that stores cloud resource usage data. Samya dis-aggregates tokens of available resources and stores fractions of these tokens across geo-distributed sites. Dis-aggregation allows sites in Samya to serve client requests

independently without the need to synchronize each update. Samya also incorporates a learning mechanism to predict the future resource demands at each site. If the predicted demand cannot be satisfied locally at a site, sites execute a synchronization protocol called *Avantan* to rebalance the available resource tokens in the system. Avantan is a novel fault-tolerant consensus protocol where sites agree on the global availability of resources that are then redistributed. After the redistribution, the sites continue to independently serve client requests. Our experiments, conducted on Google Cloud Platform, highlight that dis-aggregating data and reducing the number of synchronizations allows Samya to commit 16x to 18x more transactions than current state of the art cloud based geo-distributed databases.

3.2 Introduction

Many small and mid-sized enterprises rely on large cloud providers, such as Amazon AWS, Google GCP, and Microsoft Azure, to provide the backend infrastructure. While the cloud's *pay-per-use* strategy along with the elasticity to spawn new resources on demand has many benefits, it comes with a cost: an unexpected traffic spike can drastically increase the consumed resources, leaving the customer with a hefty bill.

To avoid such surcharges, cloud customers can set limits on the amount of resources they consume through a variety of *resource tracking services* [182]. Clients can set limits on resources such as storage capacity, number of deployable VMs, and network bandwidth. Resource tracking services within a cloud provider actively maintain data on current resource usage; this data helps enforce the limits and bill the customer accurately for their usage. A resource can be consumed only if its current usage is below the preset limit of that resource – this translates to a *read-write* transaction at the resource tracking services.

Consider an example where a large cloud provider, *ultraCloud*, has a start-up *eCommerce.com* as a customer. The start-up comprises of many teams such as clothing, electronics, etc, as shown in Figure 3.1, and the teams consume resources as indicated in the leaf nodes. The resource limit is set by an admin of *eCommerce.com* and is applicable to all teams within the organization. This type of hierarchical structure is widely used by the cloud providers to allocate resources, track the usage, and accurately bill the customer [82, 5, 154].

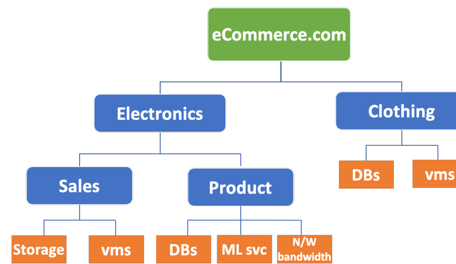

Figure 3.1: Hierarchical org structure of a cloud customer *eCommerce.com*.

The cloud provider, *ultraCloud*, tracks the number of resources *eCommerce.com* can consume. For example, each vm creation a read-write transaction in *ultraCloud*'s resource tracking service to check if the **overall** vms consumed exceeds *eCommerce.com*'s threshold. Only after this transaction succeeds can the actual physical resource be allocated. Any update to an intermediary or leaf unit (team) must percolate to the root node, *eCommerce.com*, as the cumulative resource usage by all the teams in the hierarchy is tracked at the root level. Typical update rates for a single node in the hierarchy may be in the hundreds of transactions per second, but the aggregate load on the root for a moderately sized enterprise hierarchy may easily be in thousands of transactions, causing the root node's data to become a ***hotspot***.

In a cloud setting, data – including a tracking service's data – are stored on *multiple* servers in a data center to ensure high availability and fault-tolerance. Examples of geo-distributed databases are Google's Spanner [45] and Amazon Aurora [209].

Consider the design choices of a Spanner-like database: each data item is replicated across multiple sites, one of which acts as a leader. For each update, the leader replicates the change onto a set of replicas using consensus protocols such as Paxos [122]. While this is a good choice for high availability, it aggravates the hot-spot problem in two ways: (1) *Sequential execution*: for hot-spots, where many transactions access the same data, conflicting transactions are processed by the leader sequentially; and (2) *High Latency*: each update is propagated to geographically distant sites, incurring high latency. Spanner commits a transaction with a mean latency of 17ms and a tail latency of 75ms [45]; hence for a single data item, *Spanner can commit on average 58.8 transactions per second (tps) and a tail throughput of 13.3 tps.* For a customer such as eCommerce.com (Figure 3.1), perhaps 60 tps is enough for an individual node, but for the aggregate root node with hundreds of teams in the hierarchy, this throughput value becomes problematic.

Our observation is that while geo-distributed databases are a good choice for supporting complex forms of data, they are not ideal for simple aggregate data types where the operations are mostly limited to additions or subtractions, such as maintaining resource usage data. Spanner-like solutions provide high scalability but fail to provide the high throughput necessary for hot-spot data. Based on this observation, our research objective is to *design an alternate system that manages simple data types and provides high throughput for update heavy workloads in a cloud setting.*

This question has been addressed for traditional non-cloud databases in many works such as [171, 16, 118, 79]. Escrow transactions [171] introduced the notion of concurrent transactions updating different 'chunks' of the an aggregate data, albeit in a non-distributed database. Barbara et al. [16], Kumar et al. [118], and Golubchik et al. [79] introduced the idea of partitioning aggregate data onto multiple sites allowing each site to independently update its portion of the data value (e.g., multiple sites independently selling airline tickets). The problem is made non-trivial by introducing a constraint while

updating the distributed data (e.g., not selling more airline tickets than the available seats). The solution proposed in this chapter is motivated by these works, adapted for the radically different settings of large scale geo-distributed cloud infrastructures.

If partitions of available resources are to be stored on different sites, the next logical question to ask is: how to distribute the available resources among these sites? The advancements in machine learning and deep learning techniques as well as the abundance of cloud resource demand data collected by cloud providers can aid in answering this question. In fact if resource demand can be predicted and resources can be allocated to sites accordingly, most client requests can be served locally, without incurring expensive cross-datacenter communications.

In this chapter, we propose an alternate design for geo-distributed data management systems to manage aggregate data. Specifically, we present ***Samya***[1] – a system that stores and tracks resource usage data across geo-distributed sites. Samya avoids the high latency and low throughput of Spanner-like databases by allowing a site to serve a client request locally, without the need for expensive synchronization.

Overview: To serve client requests locally while still maintaining the global resource limit, sites in Samya start with an initial allocation of available resources. We model the resource data as *tokens* (tokens of a specific resource are indistinguishable). A site can serve requests locally as long as it has locally available tokens; once it exhausts its local tokens or if it predicts an increase in resource demand that cannot be satisfied locally, the sites synchronize to *redistribute* any unused tokens in the system. We propose a novel protocol –*Avantan*[2] – to redistribute spare tokens.

Avantan is a fault-tolerant consensus protocol, in which, unlike Paxos, the value to agree upon is unknown at the start of the protocol. The sites communicate with each

[1] Samya is the Sanskrit word to equilibrium or equality.
[2] Avantan in Sanskrit means allocation.

other to share their local token values and attempt to reach agreement on the shared values. If successful, the sites use the shared values to redistribute any spare tokens. Thus, sites in Samya are constantly rebalancing the tokens among themselves based on the demand predictions, to maximize the number of client requests served with minimal latency.

Along with providing low latency, the dis-aggregation strategy of Samya increases its availability compared to Spanner-like databases. For a specific resource, Spanner becomes unavailable if a majority of the sites that store the resource information fail, whereas Samya is available as long as at least one site is available.

Other Applications of Samya: Although Samya is motivated and presented as a service that stores and tracks resource usage, it can be used as a data managing system to maintain *any* aggregate data in the cloud. Examples of applications consisting of aggregate data are: rate limiting services to manage quotas and policies; inventory management such as online shopping, car rentals, etc.; airline ticket booking; advertisement campaigns tracking; billing services; etc,. For ease of exposition, in this chapter we focus on one application: maintaining resource usage data.

The chapter is structured as: §3.3 discusses existing works related to Samya, §3.4 discusses the system and data model employed in Samya. §3.5 explains transaction executions and introduces Avantan, §3.6 presents the experimental evaluation of Samya and §3.7 concludes the chapter.

3.3 Related Work

The hotspot problem for aggregated data fields is an important practical problem studied extensively by the database community. Data partitioning is the most predominantly adopted solution for the hotspot problem, generally present in the two main forms:

(i). *Key partitioning* where data items are partitioned into different, non-overlapping sets based on their keys and the sets are stored across multiple sites; and (ii). *Value partitioning* where the same data item, irrespective of its key, is partitioned into different *values* and these values are stored across multiple sites. Since Samya is designed for a single high contention hotspot data, such as the root of an organization hierarchy, Samya adopts the value partitioning approach to store fractions of an aggregate value (i.e., available tokens) across different sites.

The idea of value partitioning has been studied extensively, starting with the seminal paper by O'Neil [171]. In [171], O'Neill introduced escrow transactions where different transactions operate on different fractions of the same data, thus allowing concurrency; this was proposed for a non-distributed database. In [118], Kumar and Stonebreaker extended transactions acquiring escrows to sites acquiring escrows. The sites serve transactions locally as long as they have non-zero escrow quantity. In [97], Harder extended the idea of escrows and introduced hierarchical escrows to reduce coordination to dynamically update escrows of multiple sites. In [117], Krishnakumar and Bernstein proposed Generalised Site Escrow to dynamically allocate parts of aggregate data (i.e., resources) to different sites using quorum locking and gossip messages.

The demarcation protocol [16] introduced by Barbara and Garica-Molina partitions an individual data value (which has a global constraint) and stores different partitions on separate machines; the protocol explains how to maintain constraints on the data when the data is distributed. In [3], Alonso and El Abbadi extend the demarcation protocol to store the value partitions across more than two sites and formalize the theory of partitioned data. In [79], Golulbchik and Thomasian introduce a token allocation system assuming that the incoming request pattern follows a Poisson distribution and tokens are allocated to different sites based on this distribution.

In essence, the above discussed works aim to partition a data item based on its value, store the partitions on multiple sites, and update them concurrently, while maintaining a global constraint. These protocols are proposed for radically different environments where typically the sites are not geo-distributed, the networks are assumed reliable, and the results presented are typically via simulations. Samya brings the basic idea – *dis-aggregate the aggregate data to increase concurrency* – from these works into the more modern context of cloud computing and geo-distributed data management systems.

A related approach that supports local operations without global synchronization is proposed by Shapiro et al. [187] in the context of Conflict-free Replicated Data Types (CRDTs). CRDTs supports conflict freedom by using eventually consistent replicas on different sites. Due to the eventual consistency guarantees and the semantics of the data types, replicas are updated locally and are synchronized with other replicas eventually. CRDTs, or rather systems that use CRDTs, differ from Samya in that the replicas of CRDTs do not dis-aggregate the value of a data item nor maintain a global and distributed constraint, which are important aspects of Samya. All the replicas of a data item in CRDT systems eventually become consistent with each other, without a notion of re-balancing the values maintained by each replica, as is performed in Samya. Another major difference between CRDT systems and Samya is CRDTs are typically commutative whereas data in Samya can be non-commutative.

Recent works have proposed key partitioning as a way to scale and improve the throughput of database systems, such as Schism [48], Horticulture [176], Clay [186], E-Store [197], and Chiller [225]. All of these works differ from Samya in two major ways: (i). they partition the data records and optimally place hot records on different sites (except Chiller [225] which places hot records on the same site) to load balance and increase performance; (ii) they optimize for a set of hot records by opting for key partitioning, whereas Samya optimizes for a single hot record by choosing value partitioning.

With the rise of the cloud paradigm, many new database designs opt for geo-distribution to provide high availability [45, 52, 209, 14, 198]. The data in these systems are key partitioned and replicated across geo-distributed sites. Google Spanner [45], Amazon Aurora [209], and CockroachDB [198] all use replication protocols such as Paxos [122] or Raft [172] to consistently replicate each update to a quorum (typically majority). While Amazon's Dynamo [52] chooses eventual consistency and is hence less stringent in replicating each update, it may suffer from inconsistent data. Recently, there has also been increasing interest in efficiently executing transactions on data that is *both* partitioned and replicated in a cloud geo-distributed environment, in works such as MDCC [116], TAPIR [227], Replicated Commit [143], G-PAC [148], and Janus [161].

In general, these approaches differ from Samya in that they employ key partitioning and aim to efficiently execute distributed transactions across these partitions, which are often also replicated. First, due to replicating each update on to a quorum of geo-distributed sites, all of the above systems are prone to hot-spot problems for update heavy and contentious workloads. Second, the design mantra common across these works is to build a general data management system that can store varied and complex forms of data. While the general approach has many benefits, it fails to take advantage of application specific data forms (such as aggregate data fields) to optimize performance. This causes applications such as cloud resource tracking services to inevitably design their own data managing systems. Samya is designed to take advantage of aggregate data to provide high performance for hot-spots without compromising availability.

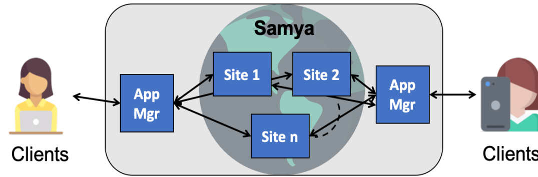

Figure 3.2: Clients interact with the data stored across sites through application managers (app mgr).

3.4 Samya Architecture

3.4.1 System Model

Samya is a niche distributed data management system that stores and maintains aggregate data specifically related to resource usage information; the data is stored across multiple geo-distributed sites. Figure 3.2 represents the system model and the client interactions with the system. Samya consists of *sites* and *application managers*.

Sites: To enhance the performance and for high availability, the aggregate data is dis-aggregated into different partitions and the partitions are stored across multiple sites, typically geo-distributed. Sites in Samya act as *data shards* that store fractions of available resources and partial usage information of a resource; and for simplicity, we assume that all sites store information about all resources. Changing this design choice to allow only some sites to store information of specific resources is fairly straightforward, and furthermore, a run-time library can provide lookup and directory services to identify the sites that store a specific resource.

Application Managers: These are stateless processes that relay the messages between a client and the sites. App managers mask the network topology and individual site availability from external clients. Since the sites storing the data and the clients accessing the data are geo-distributed, multiple geo-distributed app manager processes exist to reduce the communication latencies between clients and sites. Being stateless, app manager processes can easily scale on demand depending on the request load.

Samya assumes an underlying asynchronous communication network where messages can be delayed, dropped, or reordered. The sites and the application managers can fail by crashing but do not exhibit malicious behavior. Unless they crash, the sites and application managers execute the designated protocol correctly. Samya further assumes that a site, which stores the data, *does not crash indefinitely*; when a crashed site recovers, it reconstructs its previous state before the crash. If an application manager crashes, since app managers are stateless, a new process can be spawned easily and plugged into the system.

3.4.2 Data Model

Abstractly, we term each resource stored in Samya as an *entity*. The clients (i.e., cloud customers) acquire or release these entities and Samya tracks client actions to maintain up-to-date resource usage and resource availability information for each entity. A high privilege-user (e.g., admin of an enterprise) within a client configures a preset maximum $\mathcal{M}_e$ – the maximum limit of available tokens for entity e – and other clients (e.g., smaller organizational units in the enterprise or end users of the enterprise) can acquire or release specific quantities of the entity.

Samya maintains the following system level constraint: at no point does the system allow the clients to collectively acquire more than $\mathcal{M}_e$ tokens for an entity e.

$$0 \leq total_acquired_tokens \leq \mathcal{M}_e \tag{3.1}$$

item	***description***
id	*UUID* to identify type of resource
$TokensLeft_S$	Num. of tokens left at site S
$TokensWanted_S$	Num. of tokens site S wants

Table 3.1: State variables of an entity e maintained by each site in the system.

The state of an entity e, as maintained by each site S in the system, refers to specific details as presented in Table 3.1: *id* is a unique identity to identify the type of entity (or resource) e; $TokensLeft$ indicates the number of tokens of entity e available at site S; $TokensWanted$ indicates the number of tokens of entity e that site S needs during a redistribution.

Transactions: Clients perform 2 types of transactions:

- *acquireTokens(e, n)*: A client asks for n tokens of entity e, where n is a positive integer.
- *releaseTokens(e, m)*: A client returns m tokens of entity e back to the system, where m is a positive integer. These tokens can later be acquired by other clients. An individual client never returns more tokens than what it has acquired.

3.5 Samya

In this section we discuss how Samya efficiently manages and tracks resource usage. Samya is a highly available distributed data management system proposed as an alternative to manage resource data in geo-distributed databases. If a client consumes any resource such as creating additional VMs, then in traditional geo-replicated databases, all the replicas are updated to reflect the resource usage. Samya, on the other hand, chooses a single site to update the resource usage data, thus avoiding the cross data-center communications for each update. To cope with varying resource demands at different sites, Samya relies on learning based predictions and dynamic reallocation of resources.

3.5.1 Overview

In this section, we provide an overview of Samya's request serving approach. A site receives either an *acquireTokens(e, n)* or *releaseTokens(e, m)* request from a client, where e identifies the entity, and n, m are the number of tokens to be acquired or released. The main goal of each site in Samya is to serve as many requests locally as possible while maintaining the global constraint that the overall acquired tokens of an entity e stored across all sites never exceeds the limit $\mathcal{M}_e$. Since a *releaseTokens* request returns tokens, it never violates the global constraint and hence, can always be served locally at a site.

Meanwhile, a site may receive an acquire request with a value greater than the number of tokens available locally at that site. A site could choose to reject these requests but Samya takes an alternate approach. If a site S cannot serve an acquire request locally, it triggers a *redistribution* by communicating with other sites. The sites share the state of their tokens for entity e and redistribute any spare tokens, after which site S may acquire enough tokens to serve pending or future client requests. We term this as *reactive redistribution*: a redistribution triggered in response to a client request that could not be satisfied locally

While reactive redistributions avoid rejecting client requests merely because tokens are exhausted locally, the requests that cause redistributions incur large delays. The cloud computing literature [80, 46, 168, 56, 107, 41] has shown that resource demand typically can be predicted. We take advantage of predictable workloads to trigger *proactive redistributions* – redistributions where a site predicts if the demand is increasing and triggers a redistribution to satisfy the predicted load. This approach further minimizes the latency to serve client requests.

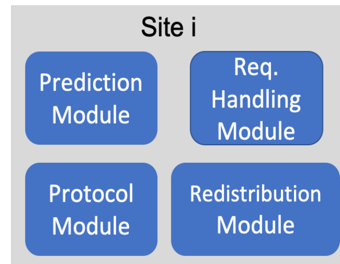

Figure 3.3: Components of each site in Samya.

Components of a Site in Samya

Each site in Samya consists of 4 components as shown in Figure 3.3.

- **Request Handling Module**: This module communicates with app managers and serves client requests locally. This module also triggers redistributions.
- **Prediction Module**: This is a learning module generated by training on existing resource demand data such that it predicts the future resource demand in terms of number of tokens.
- **Redistribution Module**: If the Request Handling module triggers a redistribution, this module calls the Protocol module to check if other sites have any spare tokens; based on the responses from other sites, this module re-allocates the spare tokens.
- **Protocol Module**: This module executes a multi-round fault-tolerant protocol that collects token information from other sites for redistribution.

Each of these modules is pluggable and can be easily replaced with an upgraded version, if and when needed.

Life-cycle of a client request

A step-wise overview of how sites in Samya serve client requests is presented in Fig. 3.4:

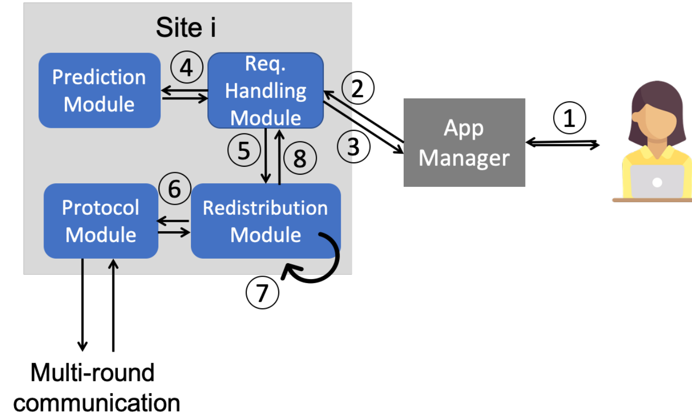

Figure 3.4: Life-cycle of a client request.

1). Client request: A client generates and sends either an *acquireTokens(e, n)* or *releaseTokens(e, m)* request, where e is the entity id, and n, m are number of tokens. This request reaches the closest app manager to the client (this can be achieved using a load balancer).

2). App Manager: Typically, the app manager relays the client request to the closest site. But if the closest site has failed or is overloaded, an app manager may relay the client request to another site. As a result, a single client's *acquireTokens* request may be sent to a site S_k whereas a *releaseTokens* request may be sent to a different site S_j. This is acceptable because sites in Samya only store the resource usage data; Samya is not responsible for the actual physical resource allocation, which is the function of higher level applications.

3). Site serving request: A site S that receives a client request attempts to serve the request locally; if successful, it updates its local token state based on the type of request:

$$TokensLeft_S = TokensLeft_S - n \tag{3.2}$$

if the client acquired n tokens; or

$$TokensLeft_S = TokensLeft_S + m \tag{3.3}$$

if the client released m tokens. The site then responds to the client, which is relayed via an app manager.

4). Demand prediction: After serving a client request, the site checks if it is close to exhausting its tokens for that entity, i.e, *TokensLeft* is below a pre-configured threshold. If so, the site predicts the demand for the near future (e.g., next 5 minutes).

5). Trigger redistribution: If the predicted value indicates a decrease in demand, the site simply continues to serve more requests. Whereas if the predicted value indicates an increase in demand that cannot be satisfied locally, the site triggers a redistribution.

6). Execute protocol: If the site triggers a redistribution, it communicates with other sites to collectively execute a fault-tolerant protocol to share with each other the state of entity e, which includes the information shown in Table 3.1.

7). Reallocate tokens: Based on the shared information, each site independently reallocates the overall spare tokens using a deterministic reallocation procedure.

8). Update tokens state: Depending on the outcome of the reallocation, the site that triggered the redistribution may acquire more tokens, upon which it updates its state of entity e and serves any pending or future requests.

Note that the above steps describe a *proactive* redistribution. Samya also supports a *reactive* redistribution triggered when a site receives a client request that cannot be served locally (due to insufficient locally available tokens). Cloud workloads typically consist of spikes and a reactive approach caters to such spiky workloads.

In the following sections, we elaborate on when a redistribution is triggered, how the redistribution protocol is executed, and once the protocol terminates, how the spare

tokens are reallocated. Table 3.2 defines the variables used in the following sections.

Symbol	Meaning
$\mathcal{M}$	Maximum Limit (of entity e)
N	Number of sites
TU_t	Tokens Used at t^{th} redistribution
TL_t	Tokens Left at t^{th} redistribution
TW_t	Tokens Wanted at t^{th} redistribution
$\mathcal{S}_t$	Total spare tokens in t^{th} redistribution
$\mathcal{R}_t$	Set of sites participating in t^{th} redistribution
$\mathcal{L}_t$	List of state variables of the sites in $\mathcal{R}_t$

Table 3.2: Variables used in the tth redistribution of an entity e at site i.

3.5.2 Triggering Redistribution

Before delving into the details of triggering a redistribution, we discuss predictability of cloud workloads. The cloud computing literature consists of many works that highlight the predictability of resource demands in the cloud, e.g., [80, 46, 168, 56, 107, 41]; they also discuss various techniques to predict resource demand. The common underlying idea is to collect a large amount of actual demand data, analyze this data to uncover any periodicity or patterns, and develop mathematical models that can learn from the past data to predict future demands.

Samya adopts a similar approach where application-specific historical resource demand data is collected to train a learning model. Once this model is trained and tuned to predict future demands, it is used as the Prediction Module (Figure 3.3). The Prediction Module is a pluggable module wherein the application developers using Samya are free to choose the best prediction technique suitable for their workload. This module can be replaced even after deployment, if a better learning approach is found or if the application workload changes. We discuss the prediction methods used in evaluating Samya in §3.6.

An *epoch* is defined as the look-ahead time duration used during prediction. This dictates how far ahead in the future to predict resource demand (e.g., 5 or 10 minutes) depending on the workload pattern. Samya triggers a redistribution in two ways.

- *Proactive redistribution*: After a site serves an *acquireTokens(e,n)* request, in a background thread, it checks if it is close to exhausting its local tokens for that entity. If so, it uses the Prediction module to predict resource demand for the next epoch. If demand is decreasing, the site continues to serve client requests. Whereas, if demand is to increase in the next epoch such that it cannot cater to the increasing demand locally, the site triggers a redistribution by updating its state of entity e:

$$TokensWanted = PredictedValue - TokensLeft \tag{3.4}$$

- *Reactive redistribution*: Since prediction techniques are rarely 100% accurate, Samya allows for *reactive* redistribution wherein a site receives an *acquireTokens(e,m)* request asking for m tokens and $m > TokensLeft$ at the site. In order to satisfy this request, the site triggers redistribution by updating its state of entity e:

$$TokensWanted = m \tag{3.5}$$

3.5.3 Executing Redistribution Protocol

Once a site decides to trigger redistribution, it executes *Avantan*: a novel fault-tolerant consensus protocol designed specifically for redistributing available resources. In this section we present two different versions of *Avantan* differing primarily in their failure assumptions and failure recoveries. Sites execute multiple instances of Avantan either sequentially or concurrently; a single execution instance is presented in this section. For each instance of redistribution, the Avantan protocol aims to reach agreement on the

list of sites participating in that instance. The protocol is designed to tolerate arbitrary crashes, message losses, and network partitions, while making the best effort in providing liveness.

The two versions of Avantan are:

- **Avantan**$[\frac{n+1}{2}]$: Requires a majority ($\frac{N}{2}+1$) of sites to be alive and communicating during protocol execution. This version of Avantan is a better choice when individual network links are highly unreliable (prone to message drops) and servers crash frequently but network partitions are infrequent. In this version all sites execute one redistribution after another.

- **Avantan**$[*]$: No requirements on majority of sites being alive to execute the protocol; it tolerates network partitions of arbitrary sizes and allows different partitions to execute redistribution concurrently. But this version is sensitive to message losses during the execution of the protocol.

The two versions also differ in their failure recovery mechanism, which will be discussed later. In developing the protocol, we follow the abstractions defined in the Consensus and Commitment (C&C) framework [148] and the protocol is motivated by the Paxos Atomic Commit (PAC) protocol proposed in [148]. Avantan abstractly consists of the following phases: the first phase executes *Leader Election* as well as *Value Construction*, the second phase makes the value *Fault-Tolerant*, and finally, the third asynchronous phase distributes the *Decision*.

We explain the two versions of Avantan with respect to redistributing the tokens of a single entity e; the protocol can be easily extended to include multiple entities. During protocol execution, sites maintain the variables defined in Table 3.3, which mainly correspond to the standard variables used in Paxos. $BallotNum$ is a tuple of the form $<num, id>$ where num is a local, monotonically increasing integer and id is site id. Ballot

Algorithm 5 Avantan[$\frac{n+1}{2}$] redistribution protocol.

Let $state_{t.i}$ be the state of entity e at site with id i during t^{th} execution of Avantan.

1: **Procedure** ELECTION-GETVALUE()

2: BallotNum ← (BallotNum.num+1, selfId)
3: InitVal ← *currState* /* With an updated *TokensWanted* */
4: Send *Election-GetValue(BallotNum)* to all

5: **Procedure** ELECTIONOK-VALUE()

6: upon receiving *Election-GetValue(bal)* from S
7: **if** bal > BallotNum
8: BallotNum ← bal
9: predictedVal ← PredictForNextEpoch()
10: **if** predictedVal > currState.TokensLeft
11: currState.TokensWanted ← predictedVal - currState.TokensLeft
12: InitVal ← *currState*
13: Send *ElectionOk-Value(BallotNum, InitVal, AcceptVal, AcceptNum, Decision)* to S

14: **Procedure** ACCEPT-VALUE()

15: **if** received *ElectionOk-Value(bal, initV, acceptV, acceptN, dec)* from majority **then**
16: **if** at least ONE response with *dec*=True **then**
17: AcceptVal ← *acceptV* of that response
18: Decision ← True
19: **else if** at least one response with $acceptV \neq \perp$ /* *dec* is True for none. */ **then**
20: AcceptVal ← *acceptV* of that response
21: **else**
22: AcceptVal ← (InitVal || all received *initV*s)
23: AcceptNum ← BallotNum
24: Send *Accept-Value(BallotNum, AcceptVal, Decision)* to all

25: **Procedure** ACCEPT-OK()

26: upon receiving *Accept-Value(bal, acceptV, acceptN, dec)* from S
27: **if** bal $\geq$ BallotNum
28: AcceptVal $\leftarrow$ acceptV
29: AcceptNum $\leftarrow$ bal
30: Decision $\leftarrow$ dec
31: Send *Accept-ok(BallotNum)* to S

32: **Procedure** DECISION()

33: **if** received *Accept-ok(bal)* from majority
34: Decision $\leftarrow$ True
35: Send *Decision(BallotNum, Decision)* to all

BallotNum	initially $< 0, s >$
InitVal	state of entity e (Table 3.1)
AcceptVal	initially $\perp$ (Null)
AcceptNum	initially $< 0, s >$
Decision	initially False

Table 3.3: Variables maintained by a site with id s during each execution of redistribution protocol for entity e.

number ensures the total ordering of different redistributions. *InitVal* is the current state value of entity e (as defined in Table 3.1) at site s when the redistribution starts. *AcceptVal* represents the list of state values of the sites participating in the redistribution and *AcceptNum* is the ballot number at which a site updates its *AcceptVal*. Finally, *Decision* indicates if the sites reached agreement on *AcceptVal* at ballot *BallotNum*.

The redistribution protocol is initiated by a site S either for proactive or reactive reasons. The different phases of the protocol are shown in Figure 3.5. Abstractly, site S attempts to become the leader and collects state values from: at least a majority of the sites in the case of Avantan[$\frac{n+1}{2}$]; and any number of sites in Avantan[$*$]. We denote the set of sites participating in the tth instance of redistribution as $\mathcal{R}_t$[3]. Site S

[3]These variables are defined in Table 3.2.

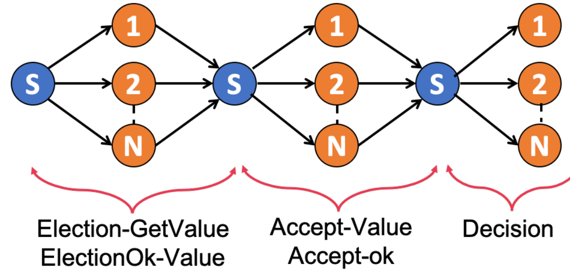

Figure 3.5: Phases of Avantan[$\frac{n+1}{2}$] protocol.

then ensures that the list of state values – as denoted by $\mathcal{L}_t$ – is fault-tolerantly stored across: a majority of sites in Avantan[$\frac{n+1}{2}$]; and the same set of sites that responded with their state values in Avantan[$*$]. S then finalizes the value $\mathcal{L}_t$ and all participating sites reallocate the tokens. Note that once a site starts participating in the redistribution protocol, it queues all the *acquireTokens* and *releaseTokens* requests from clients until the protocol terminates.

Avantan[$\frac{n+1}{2}$]

The protocol consists of 3 rounds (5 phases) as described in Algorithm 5 and shown in Figure 3.5:

- ***Election-GetValue***: In the first phase site S attempts to become the leader as well as collect the state values from other sites. Site S increments its ballot number (line 2) and sets its $InitVal$ to the current local state of entity e, i.e., all the fields of *TokensUsed, TokensLeft, and TokensWanted*. Site S then sends Election-GetValue(BallotNum) message to all sites.

- ***ElectionOk-Value***: As shown in lines 6-13, upon receiving the *Election-GetValue* message, a site C (termed as cohort to distinguish from the leader) checks if the received ballot number is greater than its current ballot number. If yes, it updates its ballot number, and it runs the Prediction Module to predict its demand for the

next epoch (line 9). If the predicted value is greater than the current number of tokens left, then site C's demand is increasing such that C cannot satisfy the increasing demand. Hence, it sets its $TokensWanted$ field (line 11) to the difference between the predicted value and its locally available tokens. C then sets its $InitVal$ to the updated state and sends ElectionOk-value(BallotNum, InitVal, AcceptVal, AcceptNum, Decision) to leader S. The *AcceptVal, AcceptNum*, and *Decision* variables are used in failure recovery; in a failure-free execution, these variables are set to the initial values as defined in Table 3.3.

- ***Accept-Value***: As shown in lines 15-25, the leader site S waits until it receives ElectionOk-Value messages from at least a majority of the sites (including itself). In a failure-free execution (failure recovery explained later), S sets $AcceptVal$ to the concatenated *InitVal*s received in the ElectionOk-Value responses (line 22), and sets $AcceptNum$ to its current ballot number. S then sends Accept-Value(BallotNum, AcceptVal, Decision) to all sites.

- ***Accept-ok***: Upon receiving the Accept-Value message from the leader, indicated in lines 27-32, a cohort C checks whether the received ballot number is at least as high as its current ballot number. If yes, it updates the $AcceptVal$, $AcceptNum$, and $Decision$ variables and sends an Accept-ok(BallotNum) message to the leader.

- ***Decision***: Finally, the leader waits for Accept-ok messages from at least a majority of sites (including self), then sets its $Decision$ variable to True, and sends the Decision(BallotNum, Decision) message to all. The protocol terminates for the leader when it sends the $Decision$ message; whereas the protocol terminates for a cohort once it receives the Decision message. The sites then reallocate the tokens using the information in $AcceptVal$ and respond to any pending client requests that were queued. The sites also reset the variables defined in Table 3.3 (except $BallotNum$)

once the protocol successfully terminates.

Failure Recovery: If the leader crashes or is network partitioned from the rest of the sites, the sites must recover in order to continue serving the clients. The failure recovery execution follows the same steps as a failure-free execution: if a site S' times-out waiting for the leader's message, S' attempts to become the new leader and terminate the protocol by sending Election-GetValue message to all the sites. As shown in Procedure Accept-Value (lines 16-28), in the received ElectionOk-Value messages, if S' receives at least one message with $Decision$ as True, this implies that the previous leader had terminated the protocol and had sent at least one Decision message before failing; so S' chooses the $AcceptVal$ received in this message (lines 18-20).

If none of the received messaged has $Decision$ as True but at least one message has a non-empty $AcceptVal$, this implies that the previous leader had received all the $InitVal$s and constructed the $AcceptVal$ and had sent Accept-Value to at least one site before failing; hence the new leader S' chooses this value as $AcceptVal$ (lines 21-22). If multiple sites respond with differing $AcceptVal$s, the new leader chooses the $AcceptVal$ corresponding to the highest $AcceptNum$. Any other case implies the previous leader had either failed to construct $AcceptVal$ or to store it on a majority before failing, and hence, S' is free to construct $AcceptVal$ based on the received $InitVal$s (line 24) (this is also the failure-free behavior). The next steps of fault-tolerantly storing the chosen value and sending the decision are the same as in failure-free executions.

Fault Tolerance: As stated in the FLP impossibility result [62], no consensus protocol can guarantee termination even with a single site failure. Following the impossibility result, Avantan[$\frac{n+1}{2}$] can block if a majority of the sites fail or are unreachable, similar to Paxos.

In spite of the blocking behaviour of Avantan[$\frac{n+1}{2}$], the availability of Samya is higher

than that of a system that executes Paxos for each transaction (e.g., Spanner). This is because the Avantan[$\frac{n+1}{2}$] protocol does not block if a majority of the sites have failed in the *first* phase of the protocol. To provide liveness, we use timeouts: if a site that wants to be a leader sends out Election-GetValue message but does not receive enough ElectionOk-Value messages within the timeout period, the site terminates the redistribution and continues to serve any client requests that can be served locally. This is acceptable since the leader failed to construct any value before aborting the redistribution.

Whereas, if the leader successfully constructed a value (after receives enough ElectionOk-Value messages) and sent Accept-Value messages to all sites but it failed to make the value fault-tolerant, i.e., it did not not receive enough Accept-Ok messages, then that site and the other live sites are blocked until a majority recover.

Theorem 1*: No two distinct values are both chosen for a given instance of Avantan[$\frac{n+1}{2}$].* The recovery mechanism of Avantan guarantees *safety* of the value – if a majority of the sites accepted the value by sending Accept-ok message, then no site will agree on a different value for that instance of redistribution. This guarantee is ensured because there exists at least one overlapping site in the two sets of majority used in the first and second phase of the protocol and any new leader learns of a value that was chosen by the previous leader through the overlapping site. This guarantee holds as long as majority of the sites are alive and reachable. □

Avantan[$*$]

Avantan[$\frac{n+1}{2}$], similar to Paxos [122] and other other consensus algorithms, is restrictive as it requires communication among a majority of sites for redistribution to succeed. If a majority of the sites are down or if the network partitions such that no partition has a majority, then the sites cannot redistribute tokens and may end up rejecting

many client requests. However, the token requirements of a site S, as represented in the $TokensWanted$ field of its state, might be satisfied by fewer than a majority of sites.

The logic of redistributing tokens among a set of sites does not impose any requirements on the minimum number of sites. Hence, we propose an alternative consensus protocol that allows *any subset of sites* to participate and ensures that all participating sites agree on the same value. We modify Avantan[$\frac{n+1}{2}$] to accommodate these new requirements.

The failure free execution of Avantan[$*$] is the same as the one presented in Algorithm 5 but with 3 major changes:

(i). The leader S that triggers the redistribution sends Election-GetValue messages to all sites. But instead of waiting for responses from a majority of sites, it waits until it receives ElectionOk-Value messages (with $TokensLeft$ field set) such that S's token requirements can be satisfied; if after a predefined amount of time, if S does not receive enough responses, it aborts the redistribution and notifies other sites and the client (for reactive redistributions). All the sites whose $InitVal$s were collected form the set $\mathcal{R}_t$ – the set of sites participating in t^{th} redistribution; in all subsequent rounds, S communicates only with the sites in the $\mathcal{R}_t$, while notifying the other sites to discard this redistribution.

(ii). If a cohort site responds with ElectionOk-Value message to one leader, it rejects all other Election-GetValue messages from concurrent leaders (even if they have higher ballot) until the former instance of Avantan[$*$] is complete. This ensures that a site participates in one instance of redistribution after another.

(iii). Rather than wait for any majority of sites to respond with Accept-ok messages (as shown in line 37), the leader S waits to receive Accept-ok from ALL the sites in $\mathcal{R}_t$ before it sends out the decision.

Different sets of sites can execute parallel redistributions but an individual site participates in one redistribution at a time.

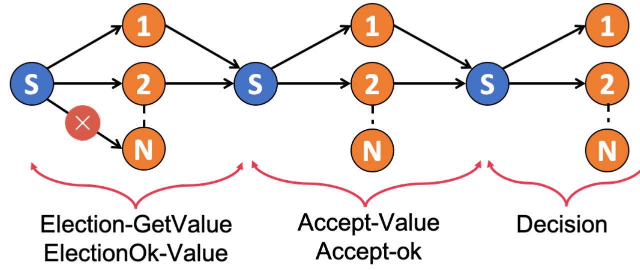

Figure 3.6: Phases of Avantan[∗] protocol.

Failure Recovery: Since Avantan[∗] does not require a majority quorum to proceed, its failure recovery differs from that of Avantan[$\frac{n+1}{2}$]. In Figure 3.6, the leader S or other participants can fail at any point during the execution of the protocol. Sites such as site N, that did not even receive the Election-GetValue message are free to participate in other redistributions. If the leader fails (crash or network partition), sites such as 1 and 2 that participated in the redistribution, must be able to recover.

A cohort site C that participated in the redistribution detects leader failure using time-outs. Upon timeout, C checks the progress of the protocol execution using the variables defined in Table 3.3. If site C's *Decision* variable is set to True, this implies the protocol had terminated and so C reallocates the tokens. If the *Decision* is not true, C decides its next action based on the value of $AcceptVal$:

i). If $AcceptVal = \perp$: This implies that C did not receive AcceptVal from the leader S before S failed; thus, C is free to abort this redistribution because the previous leader could not have proceeded to the *Decision* phase without the Accept-ok from C.

ii). If $AcceptVal \neq \perp$: This implies that the leader had chosen a value but may not have decided on it, as the leader may have failed to receive enough Accept-oks before failing. In this case, C contacts all sites in $\mathcal{R}_t$ and waits for the response from the sites in $\mathcal{R}_t$ (note that C knows all the sites in $\mathcal{R}_t$ based on list of $InitVal$s in $AcceptVal$). If any site responds with *Decision* as True, this implies the previous leader was successful

in making the value fault-tolerant but failed before sending Decision to all. Hence, site C sends the Decision message to all sites in $\mathcal{R}_t$.

Otherwise, if any site responds with $AcceptVal = \perp$, C can safely abort the redistribution (and perhaps notify other sites in $\mathcal{R}_t$) as this implies that the previous leader failed before making the constructed value fault-tolerant, and hence could have decided on it. If all sites in $\mathcal{R}_t$, except the previous leader S, respond with identical $AcceptVal$, this implies that S was successful in storing the value on all sites in $\mathcal{R}_t$ but failed before sending any Decision message. Hence, site C decides on that value, sets $Decision$ to True, and sends the Decision message. And finally, if C cannot communicate with all the other blocked sites in $\mathcal{R}_t$, C is blocked.

Fault tolerance: Similar to Avantan[$\frac{n+1}{2}$], failures during protocol execution can cause the set of sites, $\mathcal{R}_t$, participating in that execution of Avantan[$*$] to be blocked. But since Avantan[$*$] allows fewer number of sites to participate in a redistribution compared to Avantan[$\frac{n+1}{2}$], the set of sites not participating in the tth instance of Avantan[$*$] are free to serve client requests or execute another instance of redistribution. The experiments in §3.6 analyze and contrast the fault tolerance of the two versions on Avantan in a practical setting.

Theorem 2*: No two distinct values are both chosen by the set of sites participating in a given instance of Avantan[$*$].*

A value is chosen once ***all*** the sites participating in an instance of redistribution, denoted by $\mathcal{R}_t$, respond with Accept-Ok messages. In Avantan[$*$], an individual site participates in only one redistribution at a time and rejects any other concurrent redistribution request. Due to this behaviour, in a failure-free execution, sites in $\mathcal{R}_t$ have a single leader who proposes a single value. Thus, in failure-free executions, all sites participating in an instance of Avantan[$*$] agree on a single value.

If one or more sites fail while executing Avantan[∗], the recovery mechanism indicates that the live sites can either successfully terminate the redistribution or are blocked until more sites recover. Blocking implies the sites will not participate in other redistributions until the current redistribution instance is terminated, and hence, the sites in $\mathcal{R}_t$ will not choose two distinct values for t^{th} instance. □

While Avantan seems similar to Paxos, they differ in two major ways: (i) Paxos aims to reach agreement on a single, client provided value whereas Avantan collects partial values from each site and aims to reach agreement on the aggregated values, and (ii) the redistribution correctness condition (Equation 3.1) does not require a majority – a fact that is exploited in designing Avantan[∗]– which is stringent requirement of Paxos.

3.5.4 Reallocating Tokens

After a site triggers redistribution and a subset of the sites execute either versions of Avantan protocol successfully, the sites execute a deterministic procedure to reallocate the tokens. In this section, we discuss how to compute the spare tokens and the procedure to reallocate the spare tokens.

A successful execution of either versions of Avantan ensures agreement on the $AcceptVal$, which is a list of $InitVal$s, i.e.,:

$$\mathcal{L}_t =< e, TU_t, TL_t, TW_t > \forall\, i\, \epsilon\, \mathcal{R}_t \tag{3.6}$$

The reallocation logic defined in Algorithm 6 takes $\mathcal{L}_t$ as input and reallocates the available tokens among the set of sites in $\mathcal{R}_t$. *The redistribution algorithm ensures the constraint in Equation 3.1* that at no point does the token allocation count across all sites exceed the maximum limit $\mathcal{M}_e$ for a given entity e. For ease of exposition, we again focus of reallocating the tokens for a single entity e and use the variables defined in Table 3.2

to explain the algorithm.

Algorithm 6 Procedures to re-allocate spare tokens after a successful redistribution

```
1:  Procedure RedistributeTokens(L_t)
2:      S_t ← 0                         /* Spare tokens */
3:      TotalTW ← 0                     /* Total tokens wanted*/
4:      for i in R_t
5:          TotalTW ← TotalTW + L_t[i].TW_t
6:          S_t ← S_t + L_t[i].TL_t
7:      if TotalTW > S_t
8:          L_t, S_t ← RejectSomeRequests(L_t, S_t)
9:      AllocateTokens(L_t, S_t)
10: Procedure RejectSomeRequests(L_t, S_t)
11:     sortedL_t ← L_t sorted in ascending order of TW_t
12:     for i in sortedL_t
13:         sortedL_t[i].TW_t ← 0
14:         S_t ← S_t + sortedL_t[i].TL_t
15:         if TotalTW ≤ S_t
16:             break
17:     return sortedL_t, S_t
18: Procedure AllocateTokens(L_t, S_t)
19:     fori in R_t
20:         L_t[i].TokensGranted ← L_t[i].TW_t
21:         S_t ← S_t − L_t[i].TW_t
22:     for i in R_t
23:         L_t[i].TokensGranted ← L_t[i].TokensGranted + S_t / len(R_t)
```

Redistributing tokens: As defined in line 1 of Algorithm 6, the RedistributeTokens procedure takes $\mathcal{L}_t$ as input. The spare tokens and the total tokens wanted (sum of tokens specified in the $TokensWanted$ field of each site) across all sites in $\mathcal{R}_t$ are computed as shown in lines 4-6. If the spare tokens are more than the total tokens wanted, all pending client requests can be satisfied, and AllocateTokens procedure is called.

Rejecting requests: If the tokens wanted is more than the spare, some requests

must be rejected. The logic for handling this case is defined in the procedure at line 10 of Algorithm 6. We take a greedy approach to *maximise overall token usage* rather than maximise the number of requests satisfied. This is achieved by first sorting the list $\mathcal{L}_t$ in ascending order of tokens wanted (line 11); we choose ascending order since the algorithm can reject requests with least tokens wanted first. From this ascending ordered list, requests with smaller number of tokens wanted are rejected (by setting tokens wanted to 0 in line 13 and increasing the spare quantity in line 14) until the number of spare tokens exceed total tokens wanted (lines 15-16).

Allocating spare tokens: Finally, AllocateTokens (line 18) is called with updated list $\mathcal{L}_t$ and spare tokens $\mathcal{S}_t$. At this point, the redistribution satisfies all sites with non-zero tokens wanted (as the requests that cannot be satisfied are already rejected). A tokens request is granted as shown in line 20 and for each granted request, the spare quantity is updated (line 21) After satisfying all the tokens wanted requests, if any more tokens are left, they are equally distributed among all the participating sites (line 23).

3.6 Experimental Evaluation

In this section we discuss the experimental evaluation of Samya, specifically the performance of two versions of Samya where one version uses Avantan[$\frac{n+1}{2}$] and the other uses Avantan[$*$] to handle any redistributions during the experiments. Samya's performance is compared with two baselines implemented by us in Go and one open-sourced database:

i). *MultiPaxSys*: A Spanner-like geo-distributed database that executes multi-Paxos [37] for each transaction.

ii). *Demarcation/Escrow*: A value-partitioned system that captures the underlying mechanisms proposed in [16, 118, 3, 117]. Specifically, Demarcation/Escrow extends the

Demarcation protocol proposed by Barbara et al. [16] to more than 2 sites, similar to [3] by Alonso et al., and integrates the notion of site escrows used in [118] by Kumar et al. All sites start with an equal 'escrow' of an entity e (maximum limit, $\mathcal{M}_e$, divided equally among all sites), and the sites serve requests locally until they exhaust the spare escrow locally. When a request cannot be served locally at site i, i borrows escrows from one or more sites. If site i fails to borrow escrow from other sites (if they are also out of escrows), then site i rejects the client request. A stringent requirement of this baseline, inherited from [16] and [118], is it requires the network to be reliable; a message drop may lead to blocking.

(iii). *CockroachDB*: A state-of-the-art open sourced geo-distributed database that uses Raft[172] to replicate any changes to the data.

In evaluating Samya, the experiments focus on two performance aspects: *commit latency* – time taken to commit a transaction measured by the client as the time from when it sent a transaction to when it received a response to that transaction; and *throughput* – the number of transactions successfully committed per second, i.e, only the *acquireTokens* and *releaseTokens* requests that succeed are counted in throughput.

3.6.1 Resource Demand Data and Its Prediction

Samya is evaluated on a VM workload dataset published by Microsoft Azure [15]. The dataset, consisting of roughly 2 million data points, contains a representative trace of Azure's VM workload in a single geographical region collected over a month in 2017. Along with other information, it includes VM creation and deletion requests reported at discrete 5-minute intervals. A detailed description and an analysis of the dataset is published by Cortez et al. [46] where several interesting patterns of the dataset are demonstrated. A noteworthy observation among them is that the VM requests have

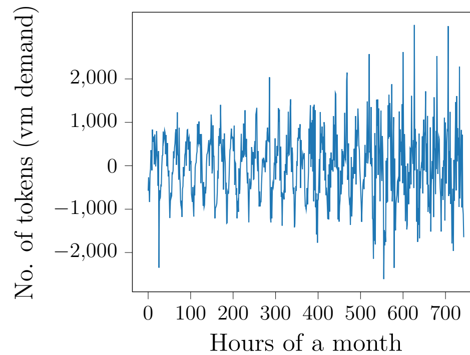

Figure 3.7: Resource demand data recorded for one month at a single region.

nearly periodic properties over time. The authors conclude that for such requests, "history is an accurate predictor of future behavior". We leverage the periodic property of the requests in the dataset to build a prediction module for Samya.

Resource Demand Prediction

The original Azure data was pre-processed such that the number of VM creations and deletions represent a demand for VMs, as depicted in Figure 3.7. The figure shows the periodically increasing and decreasing demand patterns in the data, indicating that a learning model can learn these patterns to predict future demands. Although the cloud computing literature consists of many sophisticated learning methods for resource prediction, we picked 3 simple options for resource prediction: the random walk model as the baseline model, ARIMA (autoregressive integrated moving average) as a linear regression model, and LSTM, a type of recurrent neural network, as a non-linear regression model.

To evaluate which out of the three models best predict the VM demands in Azure dataset, the original one month data was split into 80% of training data and 20% of testing data. The result of our evaluation is shown in Table 3.4. LSTM predicted the resource demands with highest accuracy, and hence, was chosen as the prediction module for Samya.

	Random Walk	ARIMA	LSTM
MAE (no. of tokens)	1212.19	609.13	259.21

Table 3.4: Mean Absolute Error (MAE) - in units of number of tokens - of resource demand prediction for three different prediction models.

Data processing

Since Samya is proposed as a solution for the *hot-spot* problem of aggregate data, the dataset used to evaluate Samya needs to have a high request-arrival-rate. To achieve this, we modified the original data's sampling interval of 5 minutes to 5 seconds. As a result, the same number of requests that arrived in a span of 5 minutes in the original dataset now arrived in a span of 5 seconds, generating a workload with high request-arrival-rate. Due to the shrinking of the sampling interval, the original duration of 30 days of the entire dataset was reduced to 12 hours.

From this 12 hours of data, we trained the LSTM prediction model with 11 hours of data, and used the last one hour (corresponding to 60 hours in the original dataset) to generate client transactions. This train and test ratio differs from the 80:20 ratio used specifically to evaluate various prediction models and to choose one among them; for real-deployment, the train and test ratio was changed to approximately 90:10.

Samya is a geo-distributed system with sites across different time zones while the Azure dataset corresponds to only a single geographical region in a single time zone. To generate the client requests at different regions, the original dataset is phase shifted based on the time difference between the regions. For example, if the demand in the original dataset peaks at 10 AM Tuesday and drops at 1 AM Wednesday, in our experiments, clients in North America generate peak demand load at 10 AM Tuesday *at the same time* as clients in Asia generate the reduced demand of 1 AM Wednesday – phase-shifted demands corresponding to the time difference between North America and Asia. The

phase shifting retains the periodicity in each region while avoiding peak demand in all regions at the same time. The clients in different regions generate respective phase-shifted transactional workloads where the VM creation and deletion requests from the dataset are transformed to *acquireTokens(VM, 1)* and *releaseTokens(VM, 1)* requests respectively.

3.6.2 Experimental Setup

The three systems, Samya, Demarcation/Escrow, and MultiPaxSys were deployed on Google Cloud Platform where each server was a general purpose `n1-standard` VM with 8 vCPUs and 30 GiB RAM. For most experiments, the VMs were placed in 5 different regions: US-West1 (US), Asia-East2 (AS), Europe-West2 (EU), Australia-Southeast1 (AU), and SouthAmerica-East1 (SA). 3 to 5 is the typical default number of replicas used in current state-of-the-art databases [53, 54]. The inter-region latency is presented in Table 3.5.

	AS	EU	AU	SA
US	131	132	161	180
AS	-	262	125	302
EU	-	-	265	218
AU	-	-	-	305

Table 3.5: Inter-region latencies in ms.

To simplify the evaluation, in the experiments, we merged the application managers and clients into a single machine. Thus, each region consisted of one VM as the client generating token acquire or release requests and another VM as the server serving client requests. In the experiments, *all five clients generated phase-shifted transactions simultaneously* and a client's requests were served by the site closest to it.

	Samya w/ Av.$[\frac{n+1}{2}]$	Samya w/ Av.[*]	Demarcation/ Escrow	MultiPaxSys	CockroachDB
90th percentile	1.40 ms	2.9 ms	3.5 ms	126.8 ms	158.7 ms
95th percentile	10.2 ms	37.3 ms	59.6 ms	172.7 ms	184.2 ms
99th percentile	65.1 ms	97.3 ms	213.9 ms	276.3 ms	351.4 ms

Table 3.6: Various latency percentiles in ms.

For MultiPaxSys and CockroachDB, since the recommendation is to place a majority of the sites in close-by regions to achieve faster replication time, we placed 3 out of 5 sites in different regions within the US, and 2 others in Asia and Europe.

All the experiments focused on entity VM and the maximum global limit, $\mathcal{M}_e$, was set to 5000, indicating that each site in Samya and Demarcation/Escrow starts with 1000 tokens. Note from Figure 3.7 that a single region's demand can go beyond 1000, ensuring that sites in Samya would require redistribution. Another implementation specific optimization was when to perform predictions: since prediction can be computationally expensive, in our experiments, a site predicts future demand only when its $TokensLeft$ value is 20% of the tokens granted value in the previous redistribution round. If the prediction indicates an increase in demand, the site triggers a proactive redistribution.

3.6.3 Latency and throughput

The first set of experiments evaluate the commit latency and throughput of the two versions of Samya, and the three baselines: Demarcation/Escrow, MultiPaxSys and CockroachDB by generating load for one hour (corresponding to 60 hours in the original dataset), creating roughly 820000 transactions. The goal of this experiment is to study the behavior of the systems over extended periods of time when the workload is highly contentious (each request either acquires or releases tokens for the same entity, VMs).

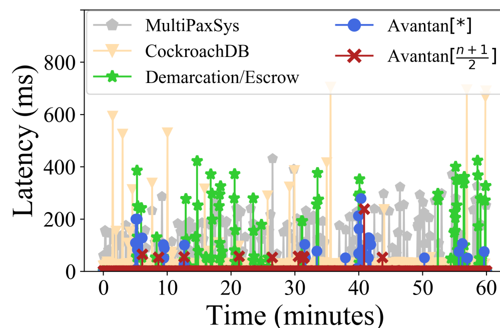

Figure 3.8: Latency of each transaction sent for an hour.

Latency: Figure 3.8 plots a sample of commit latencies of *individual transactions* sent by clients for the duration of an hour. Since each transaction in MultiPaxSys and CockroachDB executes a replication round before responding to the client, and the workload is contentious, both the systems incur significantly higher latencies compared to Samya. For Samya (both versions), most client requests are served locally at the closest site; the spikes in latencies of specific transactions indicate an ongoing redistribution during that transaction's processing. For Demarcation/Escrow, although most requests are served locally, due to the lack of prediction and an efficient escrow redistribution strategy, the peaks in resource demand causes latency peaks; hence latency of Demarcation/Escrow is higher than Samya.

Latency incurred at different percentiles for all five systems are tabulated in Table 3.6. The interesting behaviour here is the contrast in latency numbers for Avantan[$*$] and Avantan[$\frac{n+1}{2}$]. We suspected Avantan[$*$] to outperform Avantan[$\frac{n+1}{2}$], since the latter needs to wait for responses from a majority to execute a redistribution, unlike Avantan[$*$], which can proceed with any number of responses. But the latencies in Table 3.6 indicate the opposite – Avantan[$\frac{n+1}{2}$] has lower latencies than Avantan[$*$] across all percentiles.

This counter-intuitive result is explained by the difference in how the two versions construct the value during the first phase of the redistribution protocol. Avantan[$\frac{n+1}{2}$]

requires at least a majority of sites to respond with their local token values, which the leader concatenates into a single value (i.e., $AcceptVal$). This redistribution re-balances the tokens between a majority of sites. Whereas, Avantan[$*$] collects just enough responses (consisting of local token values) to satisfy its token needs, and immediately proceeds to the fault-tolerance phase. While this greedy approach may be beneficial for specific transactions, Avantan[$*$] ends up re-balancing the tokens between a small number of sites, causing more sites to trigger subsequent redistributions. Hence, in the long run, Avantan[$\frac{n+1}{2}$] is better at re-balancing the tokens and causing fewer redistributions. In the experiments, for the same client workload, Avantan[$\frac{n+1}{2}$] required 208 redistributions (proactive and reactive combined) whereas Avantan[$*$] required 792 redistributions.

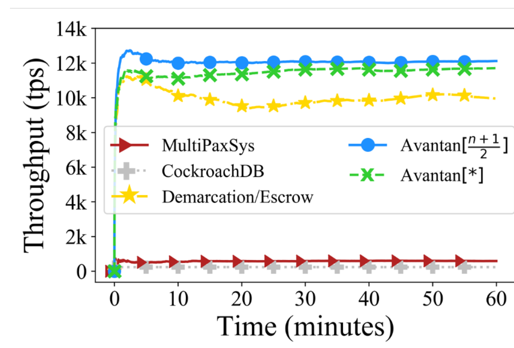

Figure 3.9: Throughput of the systems recorded for an hour.

Throughput: Figure 3.9 shows the 5-minute moving average throughput of all five systems when five clients generate concurrent requests each second and send the requests to the sites. Since MultiPaxSys and CockroachDB serve these requests sequentially (as they all update the same data entry), their throughput is roughly **16-18x** worse than Samya and **11x** worse that Demarcation/Escrow. This result highlights the *benefits of dis-aggregating an aggregate value to allow executing concurrent transactions.*

Between Demarcation/Escrow and Samya, the demand prediction and a more efficient redistribution strategy of Samya causes its throughput to be almost **1.3x** better than

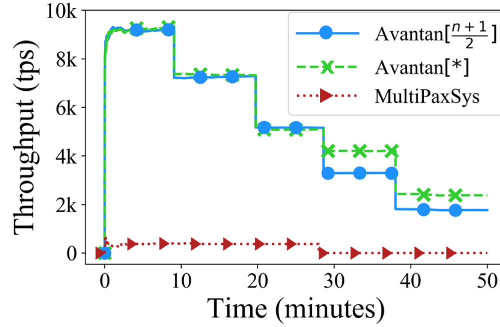

Figure 3.10: Throughput recorded when sites fail.

Demarcation/Escrow. The performance difference between Avantan[$\frac{n+1}{2}$] and Avantan[$*$] is due to the increased number of redistribution in the latter, which slows the rate with which client requests are served.

Since this experiment establishes that the performance of MultiPaxSys and CockroachDB are comparable, we use MultiPaxSys for performance comparisons in the following experiments.

3.6.4 Failure Experiments

Crash Failures

This set of experiments evaluate the Samya and MultiPaxSys when crash failures occur (Demarcation/Escrow is not evaluated in failure experiments for it requires reliable networks and hence is not fault-tolerant). The experiment starts with five regions and roughly every 10 minutes, we crash both the site and the client in a region, until only one region remains alive, while recording the throughput throughout the experiment. The results are highlighted in Figure 3.10. As indicated in the figure, once three sites crash, the throughput of MultiPaxSys drops to 0, since no transaction can be committed once a majority of the sites fail.

For the two versions of Samya, the performance is roughly the same up to 2 site failures (note that the performance is similar for both and not worse for Avantan[$*$] because in the first few minutes, the number of redistributions are low due to low resource demand in the Azure dataset; when the number of redistributions are low, the two versions perform comparably). When 3 sites fail, Avantan[$\frac{n+1}{2}$] attempts redistribution, times-out, and fails to perform any redistribution due to the failed majority. However, sites continue to serve requests that can be served locally. Meanwhile, Avantan[$*$] can successfully redistribute tokens even if only a minority of the sites are alive, thus causing its performance to be higher than Avantan[$\frac{n+1}{2}$] when failures occur.

Network Partitions

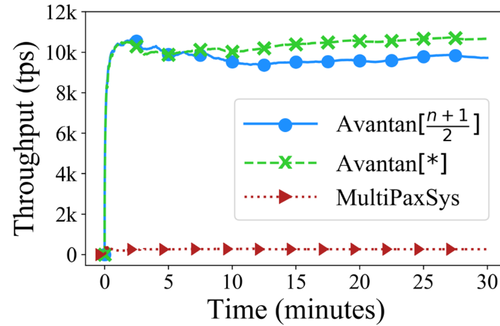

Figure 3.11: Throughput recorded during network partition.

This experiment measures the performance of Samya and MultiPaxSys during a network partition. The experiment is performed in the presence of a 3-2 network partition, i.e., one partition consists of 3 sites and the other consists of 2 sites, and clients send transactions for thirty minutes. The results are indicated in Figure 3.11. In MultiPaxSys, only the partition with 3 replicas continues to serve client requests and are up-to-data while the other two replicas are rendered stale. Its performance is significantly low compared to Samya.

For Samya, although both Avantan[$\frac{n+1}{2}$] and Avantan[$*$] start off with comparable performance, once the sites exhaust local tokens and trigger redistributions, Avantan[$*$] outperforms Avantan[$\frac{n+1}{2}$], since Avantan[$\frac{n+1}{2}$] cannot redistribute tokens in the smaller network partition whereas Avantan[$*$] can.

The two failure experiments highlight that between the two versions of redistribution strategies for Samya, Avantan[$*$] performs better in a failure prone environment, compared to Avantan[$\frac{n+1}{2}$]; but in a failure-free scenario, Avantan[$\frac{n+1}{2}$] performs better as indicated in Section 3.6.3.

One advantage of MultiPaxSys over Samya in both failure scenarios is that MultiPaxSys can allot more tokens as long as a majority of the replicas are alive, because the synchronous replication makes sure that the entire quota limit can be used. Whereas some tokens claimed tokens in Samya are lost temporarily until recovery.

3.6.5 No Constraint vs. No Redistribution

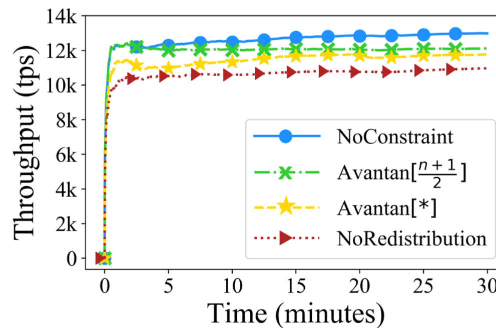

Figure 3.12: Throughput of Samya with no constraints and no redistributions vs. Samya with redistributions.

The remaining experiments focus on contrasting the two version of Avantan in Samya. In this experiment, we explore whether redistribution is worth it and the cost of redistribution on throughput. This experiment compares Samya's performance with its two baseline versions: i). *No Constraints*: there is no upper-bound on the number of resource

tokens allotted, hence every requests (acquire or release) succeeds locally at a site; ii). *No Redistribution*: there is a maximum limit constraint but once a site exhausts its local quota, it simply rejects the client request, rather than triggering a redistribution (neither proactive nor reactive). The results are shown in Figure 3.12.

Comparing the baselines: i). Samya with no constraints is the best case scenario with optimal performance, and as seen in Figure 3.12, Samya with constraints and redistributions has only 3.5-4% less throughput than the optimal throughput. ii). Samya with both versions of Avantan has about 14% higher throughput than Samya with no redistributions, i.e., 14% of the transactions would be rejected if Samya did not perform redistributions. This indicates that although executing global redistribution is expensive, the system performs better with the redistributions.

3.6.6 Proactive vs. Reactive Redistributions

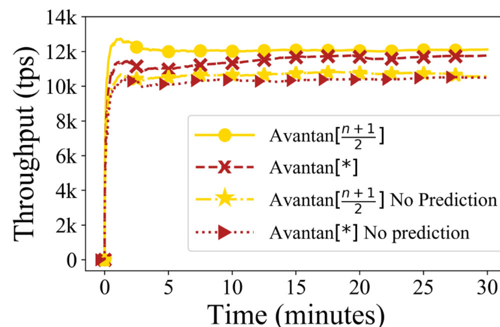

Figure 3.13: Samya's performance with and without predictions.

This experiment aims to measure the significance of predictions in Samya. Performance of four variants of Samya are measured: Avantan[$\frac{n+1}{2}$] with and without prediction, and Avantan[$*$] with and without prediction. The clients execute transactions for thirty minutes for each variant. As indicated in Figure 3.13, Samya performs about **1.4x** better with predictions (for both versions). Predictions proactively prepare a site for the incom-

ing demand and allows a site to indicate its token requirements with higher precision. This experiment highlights the advantages of using predictions in building distributed systems such as Samya.

3.6.7 Increasing number of sites

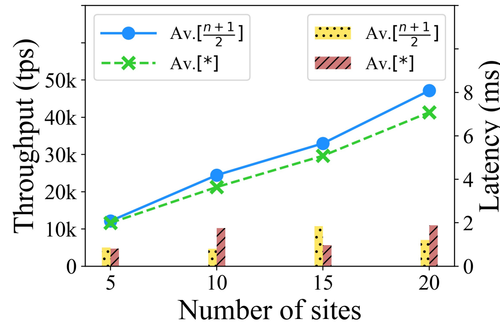

Figure 3.14: Average throughput (line graphs) and latency (bar graphs) measured for increasing number of sites.

This set of experiments evaluate the scalability of Samya by increasing the number of sites from 5 to 20, with additional sites spawned in each of the 5 regions in which previous experiments were conducted. In this experiment, for each configuration, the clients generate transactions for 10 minutes. Figure 3.14 depicts the average latency and average throughput for each configuration. As indicated in the figure, Samya shows a roughly linear increase in throughput as the number of sites increase, while keeping the average latency below 2ms for both versions of Avantan. This experiment highlights that Samya is highly scalable as more clients can concurrently acquire or release tokens when the number of sites increase.

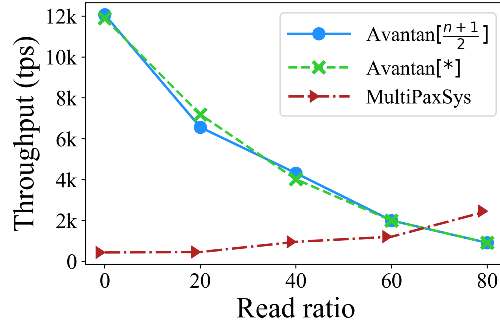

Figure 3.15: Average throughput measured with increasing ratio of read-only transactions.

3.6.8 Read-Write workload

This experiment compares the average throughput of the two versions of Avantan with that of MultiPaxSys, when the ratio of read-only transactions increases, as shown in Figure 3.15. For Avantan, when a client issues a read request to a site S, S communicates with all the other sites to learn their current token availability, aggregates the received values and responds to the client with a global snapshot of the total available tokens. For MultiPaxSys, the current available tokens is read at a single leader site. This experiment highlights the threshold at which MultiPaxSys has performance advantages over Avantan: when the read ratio increases roughly past 65%, the throughput of MultiPaxSys increases more than Avantan. Since reads are performed at a single site in MultiPaxSys and most writes are performed a single site in Samya, one would expect the crossover point to be at 50%, which is not the case. The reason is: in our experimental setup, five geo-distributed clients generate requests in parallel and for MultiPaxSys, all client requests are sent to one single leader site, which sequentially processes the requests, thus incurring high latency. Whereas for Avantan, due to the decentralised design choice, write requests are typically served locally by sites closest to the clients, in parallel. Hence, as long as an application's write load is 35% or more, it can benefit by choosing Samya.

3.6.9 Varying the maximum limit $\mathcal{M}_e$

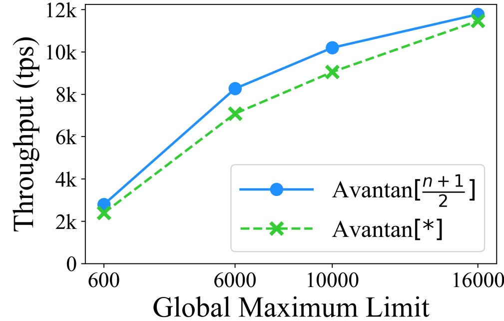

Figure 3.16: Average throughput measured with increasing maximum resource limit $\mathcal{M}_e$.

This experiment compares the average throughput of the two versions of Avantan when the maximum limit $\mathcal{M}_e$ of VM resource increases from 600 to 16000, as shown in Figure 3.16. This experiment consists of 5 sites in 5 different regions and each experimental run was executed for half an hour. From the VM demand data shown in Figure 3.7, 624 is the mean positive demand and 3118 is max positive demand at a *single* region. Hence, in this experiment we set the maximum limit from 600 to 16000. When $\mathcal{M}_e$ is set to 600, each site starts with 120 tokens each, causing roughly 1960 redistributions (predictive and reactive combined); similarly, with $\mathcal{M}_e$ set to 16000, each site starts with 3200 tokens causing no redistribution. The experiment shows that Avantan's throughput increases roughly **5x** when the maximum limit is increased from mean to max demand for the specific Azure VM demand data, thus bringing out the sensitivity of Avantan's performance with regard to the maximum resource limit.

3.6.10 Request arrival rate

This experiment measures the sensitivity of Avantan to the request arrival rate. As mentioned in Section 3.6.1, to generate a high request arrival rate, the original data's sampling interval was modified from 5 minutes (300 seconds) to 5 seconds. This experiment

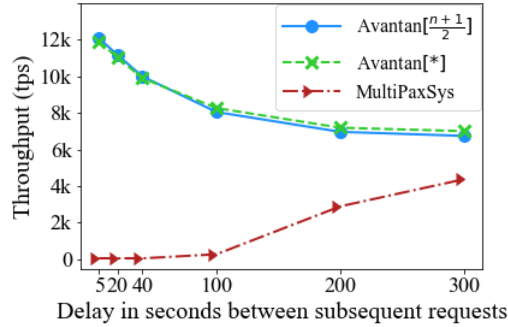

Figure 3.17: Average throughput measured with increasing delay between requests.

starts with 5 second interval and goes up to the original scale of 300 seconds. For each configuration, we measure the average throughput of a 5-minute moving average throughput. As seen in Figure 3.17, the throughput of Avantan reduces by 33% when the request arrival rate reduces by 60x (5 seconds to 300 seconds). For MultiPaxSys, we notice as increase in throughput for reduced request arrival rate as the number of contentious requests sent per second reduces, causing MultiPaxSys to commit more transactions (i.e., the rate of aborted transactions decreases). The main conclusion of this experiment is that even at the original request arrival rate, Avantan commits 43% more transactions than MultiPaxSys.

3.6.11 Limitations and Future Work

Effect of Maximum Limit $\mathcal{M}_e$: Samya's performance is inversely correlated to the number of redistributions executed by the sites (as indicated in Section 3.6.3). The higher the maximum limit, $\mathcal{M}_e$, for a resource e, the fewer the number of redistributions sites need to execute. Hence, if most resources of an application have small maximum limit value, Samya's performance benefits may reduce, as indicated in the experiment presented in Section 3.6.9.

Read-Heavy Workloads: In its current design, Samya is optimized for update heavy workloads. Samya can be easily extended to incorporate partial read transactions, i.e., transactions that read partial resource availability information stored at a single site. To obtain global resource availability information, sites in Samya execute a round of communication, unlike reading at the leader in Spanner-like systems. Therefore, Samya is a better choice if the client workload consists of at least 35% writes, as indicated in Section 3.6.8.

Global Predictions: In its current design, a site in Samya predicts future demand locally and triggers a redistribution. In addition to relying on each site's local knowledge to determine when to trigger redistribution, a global optimizer can be designed that predicts workload spike/trough at each site and triggers redistribution based on the global knowledge. This is an interesting future direction.

3.7 Conclusion

In this chapter, we propose *Samya* – a geo-distributed data management system to store aggregate data, presented as a system that specifically maintains cloud resource usage data. Samya dis-aggregates the aggregate resource usage data and stores fractions of available tokens of resources on multiple geo-distributed sites. The dis-aggregation allows concurrent updates to the hotspot data, in contrast to sequentially ordering all concurrent and contentious updates at a leader site as in traditional geo-distributed databases such as Google's Spanner. A site in Samya serves client requests independently until, based on a learning mechanism, it predicts an increase in its local resource demand that cannot be satisfied locally. This triggers a synchronization protocol *Avantan* to redistribute the available tokens, after which, sites continue to serve client requests independently. We discuss two version of Avantan where one version performs better in an infrequent

failure setting, and the other performs better when crash failures or network partitions are frequent. The experimental evaluation of Samya's performance highlights the benefit of dis-aggregation as Samya commits 16x to 18x more transactions than a Spanner-like database.

Chapter 4

Fides: Managing Data on Untrusted Infrastructure

4.1 Overview

Significant amounts of data are currently being stored and managed on third-party servers. It is impractical for many small scale enterprises to own their private datacenters, hence renting third-party servers is a viable solution for such businesses. But the increasing number of malicious attacks, both internal and external, as well as buggy software on third-party servers is causing clients to lose their trust in these external infrastructures. While small enterprises cannot avoid using external infrastructures, they need the right set of protocols to manage their data on untrusted infrastructures. In this chapter, we propose ***TFCommit***, a novel atomic commitment protocol that executes transactions on data stored across multiple untrusted servers. To our knowledge, TFCommit is the first atomic commitment protocol to execute transactions in an untrusted environment without using expensive Byzantine replication. Using TFCommit, we propose an *auditable* data management system, ***Fides***, residing completely on untrustworthy infrastructure.

As an auditable system, Fides guarantees the detection of potentially malicious failures occurring on untrusted servers using tamper-resistant logs with the support of cryptographic techniques. The experimental evaluation demonstrates the scalability and the relatively low overhead of our approach that allows executing transactions on untrusted infrastructure.

4.2 Introduction

A fundamental problem in distributed data management is to ensure the atomic and correct execution of transactions. Any transaction that updates data stored across multiple servers needs to be executed atomically, i.e., either all the operations of the transaction are executed or none of them are executed. This problem has been solved using commitment protocols, such as Two Phase Commit (2PC) [87]. Traditionally, the infrastructure, and hence the servers storing the data, were considered trustworthy. A standard assumption was that if a server failed, it would simply crash; and unless a server failed, it executed the designated protocol correctly.

The recent advent of cloud computing and the rise of blockchain systems are dramatically changing the trust assumptions about the underlying infrastructure. In a cloud environment, clients store their data on third-party servers, located on one or more data centers, and they execute transactions on the data. The servers hosted in the data centers are vulnerable to external attacks or software bugs that can potentially expose a client's critical data to a malign agent (e.g., credit details exposed in Equifax data breach [59], breaches to Amazon S3 buckets [7]). Further, a server may intentionally decide not to follow the protocol execution, either to improve its performance or for any other self-interest (e.g., the next big cyber threat is speculated to be intentional data manipulation[49]).

The increasing popularity of blockchain is also exposing the challenges of storing data

on non-trustworthy infrastructures. Applications such as supply chain management [115] execute transactions on data repositories maintained by multiple administrative domains that mutually distrust each other. Open permissionless blockchains such as Bitcoin [162] use computationally expensive mining, whereas closed permissioned blockchains such as Hyperledger Fabric [11] use byzantine consensus protocols to tolerate maliciously failing servers. Blockchains resort to expensive protocols that tolerate malicious failures because for many applications, both the underlying infrastructure and the participating entities are untrusted.

The challenge of malicious untrustworthy infrastructure has been extensively studied by the cryptographic and security communities (e.g., Pinocchio [174] that verifies outsourced computing) as well as in the distributed systems community, originally introduced by Lamport in the famous Byzantine Agreement Protocol [126]. One main motivation for the protocol was to ensure continuous service availability in Replicated State Machines even in the presence of malicious failures.

In most existing databases, the prevalent approach to tolerate malicious failures is by replicating either the whole database or the transaction manager [68, 71, 208, 230, 19]. Practical Byzantine Fault Tolerance (PBFT) [35] by Castro and Liskov has become the predominant replication protocol used in designing data management systems residing on untrusted or byzantine infrastructure. These systems provide fault-tolerance in that the system makes progress in spite of byzantine failures; the replication masks these failures and ensures that non-faulty processes always observe correct and reliable information. *Fault tolerance is guaranteed only if at most one third of the replicas are faulty [29].*

In a relatively open and heterogeneous environment knowing the number of faulty servers – let alone placing a bound on them – is unrealistic. In such settings, an alternate approach to tolerate malicious failures is *fault-detection* which can be achieved using *auditability*. Fault detection imposes no bound on the number of faulty servers – any

server can fail maliciously but the failures are always detected as they are not masked from the correct servers; detection requires only one server to be correct at any given time. To guarantee fault detection through audits, tamper-proof logs have been proposed and widely used in systems such as PeerReview [96] and CATS [219].

Motivated by the need to develop a fault-detection based data management system, we make two major propositions in this chapter. First, we develop a data management system, ***Fides***[1], consisting of untrusted servers that may suffer arbitrary failures in all the layers of a typical database, i.e., the transaction execution layer, the distributed atomic commitment layer, and the datastore layer. Second, we propose a novel atomic commit protocol –***TrustFree Commitment*** (TFCommit) – an integral component of Fides that commits distributed transactions across untrusted servers while providing auditable guarantees. To our knowledge, TFCommit is the first to solve the distributed atomic commitment problem in an untrusted infrastructure without using expensive byzantine replication protocols. Although we present Fides with TFCommit as an integral component, TFCommit can be disintegrated from Fides and used in any other design of a trust-free data management.

With detection being the focus rather than tolerance of malicious failures, Fides precisely identifies the point in the execution history at which a fault occurred, as well as the servers that acted malicious. These guarantees provide two fold benefits: i) A malicious fault by a database server is eventually detected and undeniably linked to the malicious server, and ii) A benign server can always defend itself against falsified accusations. By providing auditabiity, Fides incentivises a server not to act maliciously. Furthermore, by designing a stand-alone commit protocol, TFCommit, that leverages cryptography, we take the first step towards developing a full-fledged data management system that fully resides in untrusted infrastructures. We believe it is critical to start

[1] *Fides* is the Roman Goddess of trust and good faith.

with a strong and solid atomic commitment building block that can be expanded to include fault tolerance and other components of a transaction management hierarchy.

§4.3 provides the necessary background used in developing a trust-free data management system. §4.4 discusses the architecture, system, and failure models of Fides. §4.5 describes the auditable transaction model in Fides and also introduces TFCommit. §4.6 provides a few failure examples and their detection. Experimental evaluation of TFCommit is presented in §4.7, followed by related work in §4.8. §4.9 concludes the chapter.

4.3 Cryptographic Preliminaries

Developing a data management system built on untrusted infrastructure relies heavily on many cryptographic tools. In this section, we provide the necessary cryptographic techniques used throughout this chapter.

4.3.1 Digital Signatures

A digital signature, similar to an actual signature, authenticates messages. A public-key signature [183] consists of a public key, p_k, which is known to all participants, and a secret key, s_k, known only to the message author. The author, $\mathcal{A}$, signs message m using her secret key s_k. Given the message m and the signature, any receiver can verify whether the author $\mathcal{A}$ sent the message m by decrypting the signature using $\mathcal{A}$'s public key p_k. Public-key signature schemes are used to prevent forgery as it is computationally infeasible for author $\mathcal{B}$ to sign a message with author $\mathcal{A}$'s signature.

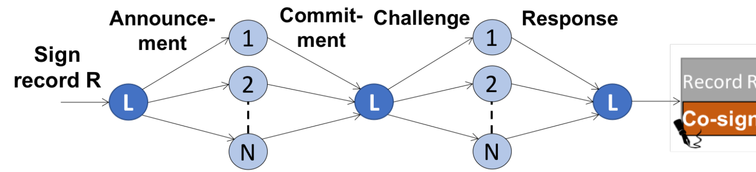

Figure 4.1: Collective Signing.

4.3.2 Collective Signing

Multisignature (multisig) is a form of digital signature that allows more than one user to sign a single record. Multisigs, such as Schnorr Multisignature [185], provide additional authenticity and security compared with single user's signature. Collective Signing (CoSi) [196], an optimization of Schnorr Multisigs, allows a *leader* to produce a record which then can be publicly validated and signed by a group of *witnesses*. CoSi requires two rounds of communication to produce a *collective signature* (co-sign) with the size and verification cost of a single signature. Figure 4.1 represents the phases of CoSi where L is the leader and $1, 2, .., N$ are the witnesses. The phases of CoSi are:

Announcement: The leader *announces* the beginning of a new round to all the witnesses and sends the record R to be collectively signed.

Commitment: Each witness, in response, picks a random secret, which is used to compute the Schnorr commitment, x_{sch}. The witness then sends the commitment to the leader.

Challenge: The leader aggregates all the commits, $X = \sum x_{sch}$ and computes a Schnorr challenge, $ch = hash(X|R)$. The leader then broadcasts the challenge to all the witnesses.

Response: Each witness validates the record before computing a Schnorr-response, r_{sch}, using the challenge and its secret key. The leader collects and aggregates all the responses to finally produce a Schnorr multisignature.

The collective signature provides a **proof** that the record is produced by the leader and that all the witnesses signed it only after a successful validation. Anyone with the public keys of all the involved servers can verify the co-sign and the verification cost is the same as verifying a single signature. An invalid record will not produce enough responses to prove the authenticity of the record. We refer to the original work [196] for a detailed discussion of the protocol.

4.3.3 Merkle Hash Tree

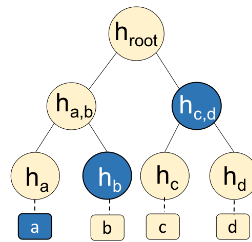

Figure 4.2: Merkle Hash Tree example.

A merkle hash tree (MHT) [152] is a binary tree with each leaf node labeled with the hash of a data item and each internal node labeled with the hash of the concatenated labels of its children. Figure 4.2 shows an example of a MHT. The hash functions, h, used in MHTs are *one way hash functions* i.e., for a given input x, $h(x) = y$, such that, given y and h, it is computationally infeasible to obtain x. The hash function h must also be collision-free, i.e., it is highly unlikely to have two distinct inputs x and z that satisfies $h(x) = h(z)$. Any such hash function can be used to construct a MHT.

Data Authentication Using MHTs: MHTs are used to authenticate a set of data values [152] by requiring the prover, say Alice, to publicly share the root of the MHT, h_{root}, whose leave form the data set. To authenticate a single data value, all that a verifier, say Bob, needs from Alice is a *Verification Object* ($\mathcal{VO}$) consisting of all the sibling nodes along the path from the data value to the root. The highlighted

nodes in Figure 4.2 form the verification object for data item a, $\mathcal{VO}(a)$, which is of size $\log_2 n$. To authenticate data item a, Alice generates the $\mathcal{VO}(a)$, and provides the value of a and $\mathcal{VO}(a)$ to Bob. Given the value of a, Bob computes $h(a)$ and uses h_b from $\mathcal{VO}(a)$ to compute $h_{a,b} = h(h(a)|h(b))$ i.e., the hash of $h(a)$ concatenated with $h(b)$. Finally, using $h_{a,b}$ and $h_{c,d}$ sent in the $\mathcal{VO}(a)$, Bob computes the root, $h_{a,b,c,d} = (h_{a,b}|h_{c,d})$. Bob then compares the computed root, $h_{a,b,c,d}$, with the root publicly shared by Alice h_{root}. Assuming the use of a collision free hash function ($h(a_1) \neq h(a_2)$ where $a_1 \neq a_2$), it would be computationally infeasible for Alice to tamper with a's value such that the h_{root} published by Alice matches the root computed by Bob using the verification object.

4.4 Fides Architecture

Fides is a data management system built on untrusted infrastructure. This section lays the premise for Fides by presenting the system model, the failure model, and the audit mechanism of Fides.

4.4.1 System Model

Fides is a distributed database of multiple servers; the data is partitioned into multiple shards and distributed on these servers (perhaps provisioned by different providers). Shards consist of a set of data items, each with a unique identifier. The system assumes neither the servers nor the clients to be trustworthy and can behave arbitrarily. Servers and clients are uniquely identifiable using their public keys and are aware of all the other servers in the system. All message exchanges (client-server or server-server) are digitally signed by the sender and verified by the receiver.

The clients interact with the data via transactions consisting of read and write operations. The data can be either single-versioned or multi-versioned with each committed

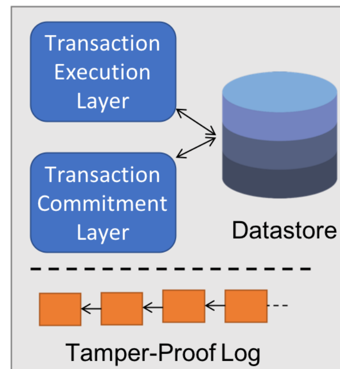

Figure 4.3: Components of a database server.

transaction generating a new version. Every data item has an associated read timestamp r_{ts} and a write timestamp w_{ts}, indicating the timestamp of the last transaction that read and wrote the item, respectively. When a transaction commits, it updates the timestamps of the accessed data items.

We choose a simplified design for a database server to minimize the potential for failure. As indicated in Figure 4.3, each database server is composed of four components: an *execution layer* to perform transactional reads and writes; a *commitment layer* to atomically (i.e., all servers either commit or abort a transaction) terminate transactions; a *datastore* where the data shards are stored; and a *tamper-proof log*.

As individual servers are not trusted, we replace the local transaction logs used in traditional protocols such as Aries [157] with a *globally replicated tamper-proof log* (this approach is inspired by blockchain). The log – a linked-list of transaction *blocks* linked using cryptographic hash pointers – guarantees immutability. Global replication of the log guarantees that even if a subset (but not all) of the servers collude to tamper the log, the transaction history is persistent.

4.4.2 Failure model

In Fides, a server that fails maliciously can behave arbitrarily i.e., send arbitrary messages, drop messages, or corrupt the data it stores. Fides assumes that each server and client is computationally bounded and is incapable of violating any cryptographic primitives such as forging digital signatures or breaking one-way hash functions – the operations that typically require brute force techniques.

Let n be the total number of servers and f the maximum number of faulty servers. Fides tolerates up to $n - 1$ faulty servers, i.e., $n > f$. To detect failures, Fides requires at least one server to be correct and failure-free (free of malicious, crash, or network partition failures) at a given time. This implies that the correct set of servers are not static and can vary over time. This failure model is motivated by Dolev and Strong's [57] protocol where the unforgeability of digital signatures allows tolerating up to n-1 failures rather than at most $\frac{1}{3}n$ malicious failures without digital signatures.

An individual server, comprising of four components as shown in Figure 4.3, can fail at one or more of the components. A fault in the *execution layer* can return incorrect values; in the *commit layer* can violate transaction atomicity; in the *datastore* can corrupt the stored data values; and in the *log* can omit or reorder the transaction history. We discuss these faults in depth in §4.5. These failures can be intentional (to gain application level benefits) or unintentional (due to software bugs or external attacks); Fides does not distinguish between the two.

A malicious client can send arbitrary messages or semantically incorrect transactions to a database server but later blame the server for updating the database inconsistently. To circumvent this, the servers store all digitally signed, unforgeable messages exchanged with clients. This message log serves as a proof against a falsified blame or when a client's transaction sends the database to a semantically inconsistent state.

4.4.3 Auditing Fides

Auditability has played a key role in building dependable distributed systems [221, 220, 96]. Fides provides auditablility: the application layer or an external auditor can audit individual servers with an intent to either detect failures or verify correct behavior.

Fides guarantees that any failure, as discussed in §4.4.2, will be detected in an offline audit. Fides focuses on failure detection rather than prevention; detection includes identifying (i) the **precise point** in transaction history where an anomaly occurred, and (ii) the exact misbehaving server(s) that is irrefutably linked to a failure.

The auditor is considered to be a powerful external entity and during each audit:

(i) The auditor gathers the tamper-proof logs from all the servers before the auditing process.

(ii) Given that at least one server is correct, from the set of logs collected from all servers, the auditor identifies the *correct* and *complete* log (how is explained in detail in §4.5.4). The auditor uses this log to audit the servers.

Optimizations such as checkpointing [113] can be used to minimize the log storage space at each server; these optimizations are orthogonal and hence not discussed further. If the audit uncovers any malicious activity, a practical solution can be to penalize the misbehaving server in legal, monetary, or other forms specific to the application. This discourages a server from acting maliciously.

4.5 Fides

In this section we present Fides: an *auditable* data management system built on untrusted infrastructure. The basic idea is to integrate crypotographic techniques such as digital signatures (public and private key encryption), collective signing, and Merkle Hash Trees (MHT) with the basic transaction execution in database systems. This integration

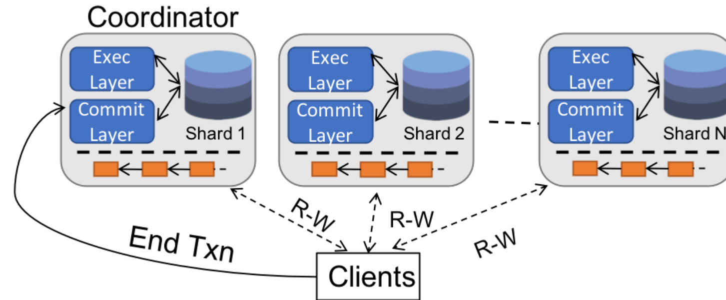

Figure 4.4: Client interactions in Fides

results in *verifiable* transaction executions in an environment where the database servers cannot be trusted.

4.5.1 Overview

Figure 4.4 illustrates the overall design of Fides. The clients read and write relevant data by directly interacting with the appropriate database partition server (this can be accomplished by linking the client application with a run-time library that provides a lookup and directory service for the database partitions). The architecture intentionally avoids the layer of front-end database servers (e.g., Transaction Managers) to coordinate the execution of transaction reads and writes as these front-end servers may themselves be vulnerable and exhibit malicious behavior by relaying incorrect reads/writes. Hence, all data-accesses are managed directly between the client and the relevant database server.

Since data-accesses are handled with minimal synchronization among concurrent activities, the burden of ensuring the correct execution of transactions occurs when a transaction is *terminated*. We use a simplified setup where one *designated* server acts as the transaction *coordinator* responsible for terminating all transactions. The coordinator is also an untrusted database server that has additional responsibilities only during the termination phase.

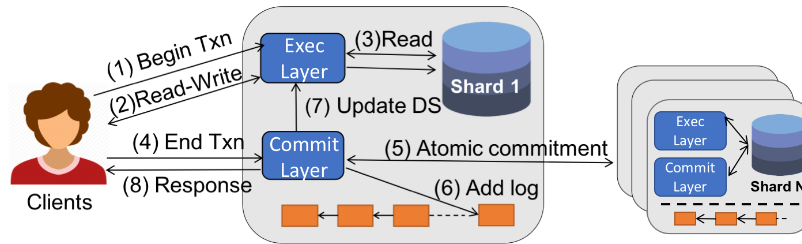

Figure 4.5: Transaction life-cycle in Fides

When a client application decides to terminate its transaction, it sends the termination request to the designated coordinator; all other database servers act as cohorts during the termination phase. For ease of exposition, we first present a termination protocol executed globally involving ***all*** database servers, irrespective of the shards accessed in that transaction. The global execution implies transactions are terminated *sequentially*. Later we relax this requirement and allow different coordinators for concurrent transactions.

The following is an overview of the client-server interaction: a typical life-cycle of a transaction as depicted in Figure 4.5.

1. Begin transaction: A client starts accessing the data by first sending a *Begin Transaction* request to all the database servers storing items read or written by the transaction.

2. Read-write request: The client then sends requests to each server indicating the data items to be read and written.

3. Read-write response: The transaction execution layer responds to a read request by fetching the data from the datastore and relaying it to the client. The write requests are buffered.

4. End Transaction: After completing data access, the client sends *End Transaction* to the coordinator which coordinates the commitment to ensure transaction correctness (i.e., serializability) and transaction atomicity (i.e., all-or-nothing property).

key	***description***
TxnId	commit timestamp of txn
R set	list of $\langle id : value, r_{ts}, w_{ts} \rangle$
W set	list of $\langle id : new_val, old_val, r_{ts}, w_{ts} \rangle$
$\sum roots$	MHT roots of shards
decision	commit or abort
h	hash of previous block
co-sign	a collective signature of participants

Table 4.1: Details stored in each block

5. Atomic commitment: The coordinator and the cohorts collectively execute the atomic commit protocol – TFCommit– and decide either to commit or abort the transaction. The commitment produces a block (i.e., an entry in the log) containing the transaction details. If the decision is commit, then the next two steps are performed.

6. Add log: All servers append, to their local copy of the log, the same block in a consistent order, thus creating a globally replicated log.

7. Update datastore: The datastore is updated based on the buffered writes, if any, along with updating the timestamps r_{ts} and w_{ts} of the data items accessed.

8. Response: The coordinator responds to the client informing whether the transaction was committed or aborted.

The log, stored as a linked-list of blocks, encompasses the transaction details essential for auditing. It is vital to understand the structure of each block before delving deeper into the transaction execution details. Every block stores the information shown in Table 4.1. Although a block can store multiple transactions, for ease of explanation, *we assume that only one transaction is stored per block*.

As indicated in Table 4.1, each transaction is identified by its commit timestamp, assigned by the client that executed this transaction. Any timestamp that supports total ordering can be used by the client – e.g., a Lamport clock with $\langle client_id : client_time \rangle$ – as long as all clients use the same timestamp generating mechanism.

A block contains the transaction read and write sets consisting of three vital pieces of information: 1) the data-item identifiers that are read/written, 2) the values of items read and the new values written; the *old_val* in the write set is populated only for blind writes, and 3) the latest read r_{ts} and write w_{ts} timestamps of those data items at the time of access (read or write).

The blocks also contain: the Merkle Hash Tree roots of the shards involved in the transaction (explained more in §4.5.2); the commit or abort transaction decision; the hash of the previous block forming a chain of blocks linked by their hashes; and finally, a collective signature of all the servers (how and why are explained in §4.5.3).

The following subsections elaborate on the functionalities of a database server in a transaction life cycle. For each functionality, we first explain the correct behavior followed by the techniques to detect malicious faults.

4.5.2 Transaction Execution

This section describes the correct mechanism for executing transactions (reads and writes) and discusses techniques to detect deviations from the expected behavior.

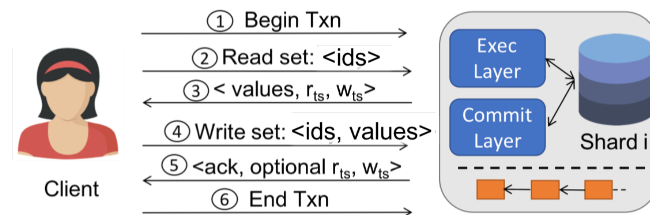

Figure 4.6: Transaction execution in Fides

Correct Behavior

Figure 4.6 depicts the client-server interactions during transaction execution. With regard to transaction execution, a correct database server is responsible for the following

actions: (i) return the values and timestamps of data-items specified in the read requests, and (ii) buffer the values of data-items updated in the transaction and if the transaction successfully commits, update the datastore based on the buffered writes. We explain how a correct server achieves these actions.

Reads and Writes: A client sends a *begin transaction* message to all the database servers storing the items read or written by the transaction. The client then sends a *Read* request consisting of the data-item ids to the respective servers. For example, if a transaction reads data item x from server $S1$ and item y from server $S2$, the client sends *Read(x)* to $S1$ and *Read*(y) to $S2$. The servers respond with the data values ***along with*** the associated read r_{ts} and write w_{ts} timestamps.

The client then sends the *Write* message with the data-item ids and their updated values to the respective servers. For example, if a transaction writes data item x in server $S1$ with value 5 and item y in server $S2$ with value 10, the client sends *Write(x,5)* to $S1$ and *Write(y,10)* to $S2$. The servers buffer these updates and respond with an acknowledgement. To support blind writes, the acknowledgement includes the old values and associated timestamps of the data-items that are being written but not read before.

After completing the data accesses, the client sends the *end transaction* request – sent only to the designated coordinator – consisting of the read and the write set: a list of data item ids, the corresponding timestamps r_{ts} and w_{ts} returned by the servers, and the values read and the new values written. The coordinator then executes TFCommit among all the servers to terminate (commit or abort) the transaction (explained in detail in §4.5.3). If all the involved servers decide to commit the transaction, each involved server constructs a Merkle Hash Tree (MHT) (§4.3.3) of its data shard with all the data items – with updated values – as the leaves of the tree and with the root node $root_{mht}$. The read and write sets and MHT roots become part of the block in the log once the transaction is committed.

Updating the datastore: If the transaction commits, the servers involved in the transaction update the data values in their datastores based on the buffered writes. The servers also update the read and write timestamps of the data items accessed in the transaction to the transaction's commit timestamp.

The data can be single-versioned or multi-versioned. For multi-versioned data, when a transaction commits, a correct server additionally creates a new version of the data items accessed in the transaction *while maintaining the older versions*. Although an application using Fides can choose between single-versioned or multi-versioned data, multi-versioned data can provide ***recoverability***. If a failure occurs, the data can be reset to the last sanitized version and the application can resume execution from there.

Detecting Malicious Behavior

With regard to transaction execution, a server may misbehave by: (i) returning inconsistent values of data-items specified in the read requests; and (ii) buffering incorrect values of data-items updated in the transaction or updating the datastore incorrectly.

(i) Incorrect Reads: All faults in Fides are detected by an auditor during an audit. As mentioned in §4.4.3, during an audit, the auditor collects the log from all servers and constructs the correct and complete log.

To detect an incorrect read value returned by a malicious server, the auditor must know the expected value of the data-item. The read and write sets in each log entry contains the information on the updated value of a written item and the read value of a read item. Note that in our simplifying assumption (which will be relaxed later), each block contains *only one* transaction and the transactions are committed sequentially with the log reflecting this sequential order. By traversing the log, at each entry, the auditor knows the most recent values of a given data item. We leverage this to identify incorrectly returned values.

Lemma 1: The auditor detects an incorrect value returned for a data item by a malicious server.

Proof: Consider a transaction T_i that committed at timestamp ts_i and stored in the log at block b_i. Assume transaction T_i read an item x and updated it. Let b_j be the first block after b_i to access the same data item x – where $j > i$, indicating that transaction T_j in b_j committed ***after*** the transaction T_i in b_i. The read value of x in b_j must reflect the value written in b_i; if the values differ, an anamoly is detected. □

(ii) Incorrect Writes: The effect of incorrectly buffering a write or incorrectly updating the datastore is the same: the datastore ends up in an inconsistent state. The definition of incorrect datastore depends on the type of data: for single versioned data, the latest state of data (data values and timestamps) in the datastore is incorrect; for multi-versioned data, one or more versions of the data are incorrect. We discuss techniques to detect incorrect datastore for both types of data.

To detect an inconsistent datastore, we use the data authentication technique proposed by Merkle [152] discussed in §4.3.3. To use this technique, the auditor requires the read and written values in each transaction and the resultant Merkle Hash Tree (MHT) root – all pieces of information stored within each block.

Multi-versioned data: For multi-versioned data, the audit policy can involve auditing a single version chosen arbitrarily or exhaustively auditing all versions starting from either the first version (block 0) or the latest version. We explain auditing a single version, which can easily be extended to exhaustively auditing all versions.

Let T_i be a transaction committed at timestamp ts that read and wrote data item x stored in server S_k. Assume the auditor audits server S_k at version ts. Once the auditor notifies the server about the audit, the server constructs the Merkle Hash Tree with the data at version ts as the leaves; S_k then shares the *Verification Object* $\mathcal{VO}$– consisting

of all the sibling nodes along the path from the data x to the root – with the auditor.

The log entry corresponding to transaction T_i stores the value read for item x and the new value written. The auditor uses (i) the $\mathcal{VO}$ sent by S_k, and (ii) the hash of x's value stored in the write set of the log, to compute the expected MHT root for the data in S_k (discussed in §4.3.3). The auditor then compares the computed root with the one stored in the log. A mismatch indicates that the data at version ts is incorrect.

Single-versioned data: For single versioned data, the correctness is only with respect to the latest state of the data. Hence, rather than using an arbitrary block to obtain the MHT root of server S_k, the auditor uses the latest block in the log that accessed the data in S_k to obtain the latest MHT root. The other steps are similar to multi-versioned data: the auditor fetches the $\mathcal{VO}$ based on the latest state of S_k and recomputes the MHT root to compare the root stored in the log.

Lemma 2: The auditor detects an inconsistent datastore. For multi-versioned data, the auditor detects the precise version at which the datastore became inconsistent.

Proof: Detection is guaranteed since Merkle Hash Trees (MHT) use collision-free hash functions (i.e., $h(x) \neq h(y)$ where $x \neq y$), and a malicious server cannot update a data value such that the MHT root stored in the block matches the root computed by the auditor using the verification object sent by the server. For multi-versioned datastores, the auditor identifies the precise version at which data corruption occurred by systematically authenticating all blocks in the log until a version with mismatching MHT roots is detected. □

4.5.3 Transaction Commitment

This section describes how transactions are terminated in Fides and presents a novel distributed atomic commitment protocol – ***TrustFree Commit*** (TFCommit) – that

handles malicious failures. This section also discusses techniques to detect failures if a server deviates from the expected behavior. With regard to transaction commitment, a correct database server is responsible for the following actions: (i) Ensure transaction isolation (i.e., strict serializability); (ii) Ensure atomicity – either all servers commit the transaction or no servers commit the transaction; and (iii) Ensure *verifiable* atomicity.

Correct Behavior

Transaction Isolation: Transaction isolation determines how the impact of one transaction is perceived by the other transactions. In Fides, even though multiple transactions can execute concurrently, Fides provides serializable executions in which concurrent transactions seem to execute in sequence. To do so, servers in Fides abort a transaction if it cannot be serialized with already committed transactions in the log. The read r_{ts} and write w_{ts} timestamps associated with each data item is used to detect non-serializable transactions. The latest timestamps can be obtained from either the datastore or the transaction log. Similar to timestamp based optimistic concurrency control mechanism, at commit time, a server checks if the data accessed in the terminating transaction has been updated since they were read. If yes, the server chooses to abort the transaction.

Atomicity and Verifiablity: Consider a traditional atomic commit protocol that provides atomicity: Two Phase Commit (2PC) [87]. 2PC guarantees atomicity provided servers are benign and trustworthy. It is a centralized protocol where one server acts as a coordinator and the others act as cohorts. To terminate a transaction, the coordinator collects commit or abort votes from all cohorts, and decides to commit the transaction *only if* all the cohorts choose to commit, and otherwise decides to abort. The decision is then asynchronously sent to the client and the cohorts. 2PC is sufficient to ensure atomicity if servers are trustworthy; but in untrusted environments, 2PC is inadequate as a cohort or the coordinator may maliciously lie about the decision. We need to develop

an atomic commitment protocol that can overcome such malicious behaviour.

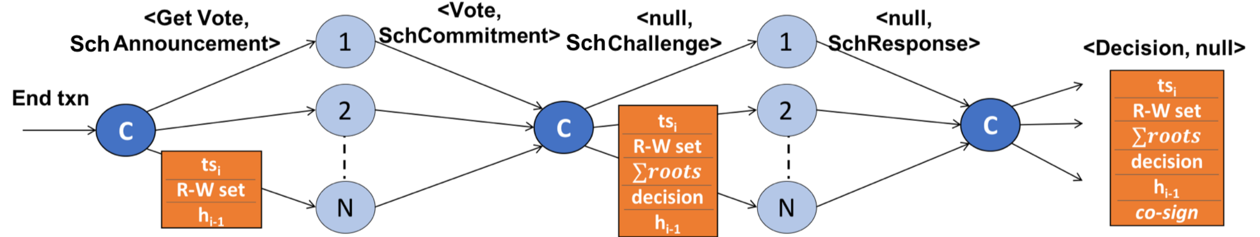

Figure 4.7: Different phases and block generation progress made in each phase of TFCommit

To make 2PC trust-free, we combine 2PC with a multi-signature scheme, Collective Signing or CoSi (§4.3.2): a two-round protocol where a set of processes collectively sign a given record using their private keys and random secrets. CoSi guarantees that a record (or in our case block) produced by a leader (or coordinator) is validated and signed by all the witnesses (or cohorts) and that *if any of the involved processes lied in any of the phases, the resulting signature will be incorrect.* A signature is bound to a single record; any process with the public keys of all the processes can verify whether the signature is valid and *corresponds* to that record.

We propose a novel approach of integrating 2PC with CoSi to achieve the atomicity properties of 2PC *and* the verifiable properties of CoSi. The basic idea is that the coordinator, similar to 2PC, collects commit or abort votes from the cohorts, forms a decision, and encapsulates the transaction details including the decision in a block. The coordinator then sends the block to be verified and collectively signed by the cohorts. An incorrect block (either with inaccurate transaction details or wrong decision) produced by a malicious coordinator will not be accepted by correct servers, thus resulting in an invalid signature that can be easily verified by an auditor.

A successful round of TFCommit produces a block to be appended to the log *in a consistent order* by all servers. For ease of exposition, this section presents TFCommit

with two main assumptions: (i) the transactions are committed sequentially to avoid forks in the log; and (ii) all servers participate in transaction termination – even the servers that did not partake in transaction execution – to have identical block order in their logs. In §4.5.6 we relax these assumptions and discuss various techniques to scale TFCommit.

Recall from Table 4.1 all the details stored in each block. Once a block is cosigned and logged by all servers, it is immutable; hence, all the details must be filled in during different phases of TFCommit. However, to ensure atomicity and verifiability of TFCommit, we only need the transaction id, its decision, and the co-sign. Other details such as the *Read* and *Write* sets, Merkle Tree roots, and hashes are necessary to detect other failures including isolation violation and data corruption.

The protocol:

A client, $\mathcal{A}$, upon finishing transaction execution, sends a signed $\mu = \langle end_transaction(T_{id}, ts_i, R\ set\text{-}Wset) \rangle_{\sigma\mathcal{A}}$ request to the coordinator, where T_{id} is a unique transaction id and ts_i is a client-assigned commit timestamp of the transaction. The request also includes $R\ set\text{-}Wset$: the read and write sets consisting of data item ids, values read and new values written, r_{ts}, and w_{ts}. The servers ignore any *end transaction* request with a timestamp lower than the latest committed timestamp.

TFCommit is a 3-round protocol involving 5 phases of communication as shown in Figure 4.7. Since TFCommit merges 2PC with CoSi, we indicate each phase by a mapping of <2PC phase, CoSi phase>. Figure 4.7 shows the phases as well as the progress made in constructing the block at each phase. The phases of TFCommit are:

1) <GetVote, SchAnnouncement>: Upon receiving the $\mu = \langle end_transaction(T_i, ts_i, R\ set\text{-}Wset) \rangle_{\sigma\mathcal{A}}$ request from the client, to commit transaction T_i, the coordinator $\mathcal{C}$ prepares a partially filled block, $b_i = [ts_i, Rset\ \text{-}\ Wset, h_{i-1}]$, containing the commit

timestamp, read and write sets, and hash of the previous block. $\mathcal{C}$ then encapsulates the signed client request μ and sends the $\langle get_vote(b_i, \mu)\rangle_{\sigma C}$ message to all the cohorts.

2) <Vote, SchCommitment>: Every cohort $\mathcal{H}$ verifies both the get_vote message and the encapsulated client request, and computes the Schnorr-commitment (x_{sch}) for CoSi. Then, *only the cohorts that are part of the transaction*, perform the following actions. A cohort involved in the transaction locally decides whether to commit or abort the transaction. If the cohort locally decides to commit, then it constructs a Merkle Hash Tree (MHT) (§4.3.3) of its shard with all the data items as leaves of the MHT and with the root node $root_{mht}$. The MHT reflects all the updates in T_i assuming that T_i be committed; since MHT computation is done in memory, the datastore is unaffected if T_i eventually aborts. (The MHT root is required for datastore authentication, as explained in §4.5.2.) The involved cohorts then send $\langle vote(decision, root_{mht}, x_{sch})\rangle_{\sigma\mathcal{H}}$ whereas the cohorts not part of the transaction send $\langle vote(x_{sch})\rangle_{\sigma\mathcal{H}}$ to the coordinator. As the coordinator is also involved in co-signing, it produces the appropriate vote message.

3) <null, SchChallenge>: In this phase, the coordinator $\mathcal{C}$ collects all the cohort responses and checks if any cohort (or itself) involved in the transaction decided to abort. If none, it chooses commit, otherwise abort. It then aggregates all the MHT roots of the involved cohorts ($roots = \sum root_{mht}$), and fills the roots field in the block b_i along with the decision field. If any involved cohorts chose abort, the respective roots will be missing in the block. Finally, the coordinator aggregates the Schnorr-commitments $X_{sch} = \sum x_{sch}$ from all the servers and computes the Schnorr-challenge by concatenating and hashing X_{sch} with b_i *i.e.*, $ch = h(X_{sch}||b_i)$. The coordinator then sends $\langle challenge(ch, X_{sch}, b_i)\rangle_{\sigma C}$ to all cohorts.

4) <null, SchResponse>: In this phase, every cohort, $\mathcal{H}$, checks if the decision within the block b_i is abort, and if so, b_i should have some missing roots; if the decision is

commit, b_i should have all the roots from the involved servers. Every involved cohort that sent the MHT root in the *vote* phase verifies if its corresponding root in the block is the same as the one it sent. Cohorts also verify whether a potentially malicious coordinator computed the challenge, ch, correctly by hashing the concatenated X_{sch} and b_i, both of which were sent in the challenge message. A cohort then computes the Schnorr-response r_i using its secret key and the challenge ch, and sends $\langle response(r_i) \rangle_{\sigma\mathcal{H}}$ to the coordinator.

5) <Decision, null>: The coordinator collects all the Schnorr-responses and aggregates them, $R_{sch} = \sum r_{sch}$, to form the collective signature represented by $\langle ch, R_{sch} \rangle$. Intuitively, the challenge ch is computed using the block; and the Schnorr-response R_{sch} requires the private keys of the servers, thus the signature binds the block with the public keys of the servers. The coordinator then updates the *co-sign* field in the block and sends the finalized block to the client and the cohorts. If the decision is commit, all servers append block b_i to their log and update their respective datastores.

The client, with the public keys of all the servers, verifies the co-sign before accepting the decision – even an aborted transaction must be signed by all the servers. If the verification fails, the client detects an anomaly and triggers an audit, which may halt the progress in the system.

TFCommit, similar to 2PC, can be blocking if either the coordinator or any cohort fails (crash or malicious). TFCommit can be made non-blocking by adding another phase that makes the chosen value available, as in the case of Three Phase Commit [189]; we leave this extension for future work.

Detecting Malicious Behavior

A correct execution of TFCommit ensures serializable transaction isolation, atomicity, and verifiable commitment. However, a malicious server can (i) violate the isolation guarantees by committing non-serializable transactions; (ii) a malicious coordinator can break atomicity by convincing some servers to commit a transactions and others to abort; or (iii) a server can send wrong cryptographic values during co-signing to violate verifiability.

Lemma 3: The auditor detects serializablity violation.

Proof: Transaction execution is based on executing read and write operations in the timestamp order. The transactions are ordered based on the timestamps, which are monotonically increasing. If a transaction has done a conflicting access inconsistent with the timestamp order, it leads to one of the following conflicts: 1) RW-conflict: a transaction with a smaller timestamp read a data-item with a larger timestamp; 2) WW-conflict: a transaction with a smaller timestamp wrote a data-item that was already updated with a larger timestamp; 3) WR-conflict: a transaction with a smaller timestamp wrote a data-item after it was read by a transaction with a larger timestamp. For each transaction audited, the auditor verifies if any of the above violations exist, and if so, the auditor detects the server responsible for the violation to be misbehaving. This is equivalent to verifying that no cycle exists in the Serialization Graph of the transactions being audited. □

Lemma 4: The auditor or a correct server detects incorrect cryptographic values for CoSi sent by a malicious server – which hampers verifiablity of TFCommit.

Proof: If any server sends an incorrect cryptographic value used for co-signing, this results in an invalid signature, and the original work CoSi [196] guarantees identifying the precise server that computed the crytographic values incorrectly. Since TFCommit

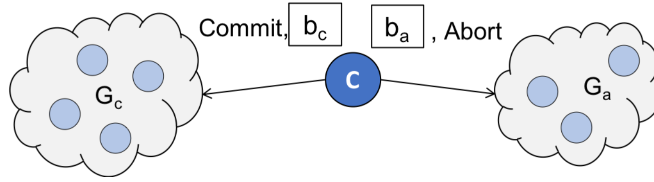

Figure 4.8: Atomicity violation of TFCommit

incorporates CoSi, it inherits this guarantee from CoSi. Intuitively, in the *schResponse* phase, the coordinator can identify if the signature is invalid, in which case, it can check partial signatures produced by excluding one server at time and detect the precise server without which the signature is valid. The coordinator is incentivised to perform this rigorous check because if the signature is invalid, the auditor suspects the coordinator for producing an incorrect block. We refer to the original work [196] that discusses the proof in depth. □

Lemma 5: The auditor or a correct server detect atomicity violation of TFCommit.

Proof: Recall that the coordinator $\mathcal{C}$ collects votes in phase two of TFCommit, forms the decision, and sends the partial block containing the decision in the *challenge* message. Consider Figure 4.8 where a malicious coordinator sends block b_c with commit decision to group G_c and block b_a with abort decision to group G_a. More precisely, the coordinator sends $\langle challenge(ch, X_{sch}, b_c)\rangle_{\sigma C}$ to G_c (X_{sch} is the aggregated Schnorr-commits) and $\langle challenge(ch, X_{sch}, b_a)\rangle_{\sigma C}$ to G_a. Since the decision is part of the block, the two blocks b_c and b_a have to be different if the coordinator violates atomicity. But with respect to the challenge ch, there are two possibilities, both producing invalid signatures:

- *Case 1*: Coordinator sends the same challenge ch computed using block b_c (or b_a) to both groups.

Any correct server in the group G_a will recompute the challenge using the block it received, b_a, and immediately recognize that the challenge sent by the coordinator does not correspond to the block b_a. (Alternatively, if the coordinator used b_a to compute the

challenge ch, then servers in G_c will detect the anomaly.) Even if the servers in one group, say G_a, collude with the coordinator and do not expose the anomaly, the challenge ch corresponds only to block b_c. The auditor, while auditing a server in group G_a, detects that the co-sign in block b_a is invalid as it does not correspond to that block.

- *Case 2*: Coordinator sends the challenge ch computed using block b_c to group G_c and the challenge ch' computed using block b_a to group G_a.

In the final step of TFCommit, the servers in group G_c will use ch to compute the Schnorr-response, whereas the servers in group G_a will use ch' to compute the Schnorr-response. Given that the final collective signature can be tied only to a single block, the co-sign does not correspond to either b_c or b_a, hence producing a wrong signature. □

The coordinator or a cohort **can never force** all servers to commit if at least one server decides to abort a transaction. For committed transaction, the transaction block must contain MHT roots from all the involved servers; for aborted transactions, the block should have at least one MHT root missing. Assume a server S_b chooses abort and hence, does not send its MHT root. If the coordinator produces a fake root for server S_b, the server will detect it in the *schResponse* phase. And in case server S_b colludes with the coordinator by either not exposing the fake root or by producing a fake root itself, the datastore verification (discussed in Section 4.5.2), which uses MHT roots, will fail for server S_b. An involved server (coordinator or cohort) can only force an abort on all servers by choosing to abort the transaction, which is tolerable as the decision will be consistent across all servers and will not violate the atomicity of TFCommit.

4.5.4 Transaction Logging

The transaction log in Fides is a tamper-proof, globally replicated log. When a transaction commits after a successful round of TFCommit, all servers append the newly

produced block to their logs.

Detecting Malicious Behavior: One or more faulty servers can collude (but not all at once) to (i) tamper an arbitrary block, (ii) reorder the blocks, or (iii) omit the tail of the log (last few blocks). The auditor collects logs from all the servers and uses the collective signature stored in each block to detect an incorrect log.

Lemma 6: Given a set of logs collected from all servers, the auditor detects all incorrect logs – logs with arbitrary blocks that are modified or logs with reordered blocks.

Proof: The collective signature in each block prevents a malicious server from manipulating that block once it is appended to the log. The signature is tied specifically to one block and if the contents of the block are manipulated, the signature verification will fail. One or more malicious servers cannot tamper with an arbitrary block successfully without the cooperation of *all* the servers. And since the hash of the previous block is part of a log entry, unless *all* the servers collude, the blocks cannot be successfully re-ordered. □

Lemma 7: Given a set of logs collected from all servers, the auditor detects all incomplete logs – logs with missing tail entries.

Proof: A subset of servers cannot successfully modify arbitrary blocks in the log (proof in Lemma 6) but they can omit the tail of the log. During an audit, the auditor gathers the logs from all the servers. At least one correct server exists with the complete log – which can easily be verified for correctness by validating the collective signature and hash pointer in each block. The auditor uses this complete and verified log to detect that one or more servers store an incomplete log. □

4.5.5 Correctness of Fides

Definition 1: *Verifiable ACID properties*

In transaction processing, ACID refers to the four key components of a transaction:
i) Atomicity: A transaction is an atomic unit in that either all operations are executed or none.
ii) Consistency: Data is in a consistent state before and after a transaction executes.
iii) Isolaiton: When transactions are executed concurrently, isolation ensures that the transactions seem to have executed sequentially.
iv) Durability: If a transaction commits, its updates are persistent even in the presence of failures.

We define *v-ACID* as the ACID properties that can be verified. *v-ACID* indicates that a database system provides verifiable evidence that the ACID guarantees are upheld. This definition is useful when individual database servers are untrusted and may violate ACID – in which case the system must allow verifying and detecting the violations.

Theorem 1*: Fides provides Verifiable ACID guarantees.*

Proof: Fides guarantees that an external auditor can verify if the database servers provide ACID guarantees or not.

The first step in the verification is for the auditor to obtain a *correct* and *complete* log. Given the assumption that at least one server is correct at a given time, Lemmas 6 and 7 prove that during an audit, the auditor always identifies the correct and complete log.

Lemma 5 proves that *Atomicity* violation is verifiable; Lemma 2 proves that the auditor verifies if the effect of a transaction resulted in an inconsistent database when a server buffers inconsistent writes, i.e., verifiable *Consistency*; Lemma 3 proves that the *Isolation* guarantee which ensures serializable transaction execution is verifiable; and finally, Lemmas 1 and 2 verify if the effects of committed transactions are *Durable.* Hence, an auditor verifies whether the servers in Fides uphold ACID properties.

Note that multiple ACID violations can exist in the transaction execution. Since the log is sequential, the auditor identifies the first occurrence of any of these violations and the blocks after that need not be audited since everything following that violation can be incorrect and hence irrelevant to a correct execution. □

4.5.6 Scaling TFCommit protocol

The TFCommit protocol discussed in §4.5.3 makes simplifying assumptions that each block contains a single transaction and a globally designated coordinator terminates all transactions which requires participation from all servers. This makes TFCommit expensive as any server not involved in a transaction must also participate in its termination. In this section we provide an intuitive overview of how to scale TFCommit.

To scale TFCommit, two aspects can be enhanced: (i) Allow multiple transactions to commit simultaneously by storing multiple transactions in a block, and (ii) Reduce the number of servers participating in transaction termination to only the servers involved in that transaction.

Extending each block to contain multiple transactions is straight-forward. The coordinator collects and inserts a set of *non-conflicting* client generated transactions and orders them within a single block at the start of TFCommit. Once the protocol begins, the coordinator or any other server cannot re-order the transactions within the block (the argument is similar to Lemma 4). This technique allows each execution of TFCommit to commit multiple transactions. In our evaluations in §4.7, we store multiple transactions in each block.

To reduce the number of servers participating in transaction termination, servers are divided into small dynamic *groups*. The servers accessed by a transaction forms one group, in which one server acts as the coordinator to terminate that transaction (instead

of one globally designated coordinator). Each group executes TFCommit internally and upon a successful execution, the coordinators of each group publish the block to all other groups. The problem with such a solution is in deciding the order of blocks *across* groups such that all the servers maintain a consistently ordered transaction log.

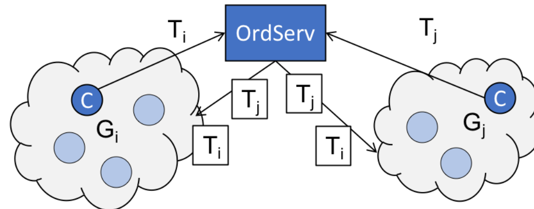

Figure 4.9: Scaling TFCommit.

There are multiple ways to solve the ordering problem. Figure 4.9 depicts a scalable solution that abstracts the ordering of blocks as a service (OrdServ). The figure shows two groups of servers G_i and G_j, each accessed by transactions T_i and T_j respectively. The OrdServ component is responsible for atomically broadcasting a single stream of blocks, each generated by TFCommit executed in different groups of servers. OrdServ can use a byzantine consensus protocol such as PBFT [35] among the coordinators to consistently order blocks; or it can be an off-the-shelf application such as Apache Kafka, used to provide ordering service in a recent work, Veritas [2]. OrdServ is also responsible for chaining the blocks i.e., the coordinators of the groups do not fill in the hash of previous block, rather it is filled by the OrdServ. There are two possible scenarios regarding the groups:

- $G_i \cap G_j = \varnothing$: If any two groups of servers have no overlapping server, there is no dependency between the two blocks of transactions T_i and T_j, and OrdServ can order them in any way and broadcast a consistent order.
- $G_i \cap G_j \neq \varnothing$: If any two groups have a non-empty intersection, then transactions T_i and T_j may have a dependency order (e.g., T_j wrote a data item after T_i read it);

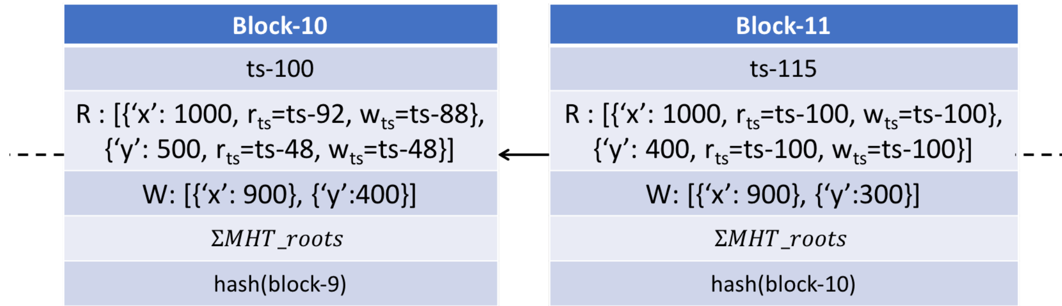

Figure 4.10: Isolation guarantee violation example.

the OrdServ should ensure that the transaction log reflects this dependency between the published blocks.

Although there is flexibility in choosing OrdServ, it is important to choose a solution that maintains local transaction order (within a group) across the globally replicated log. Solutions such a ParBlock [10] track the transaction dependency order and maintains that order while publishing blocks. We plan to integrate ParBlock with TFCommit as future work.

4.6 Failure Examples

In this section we discuss various malicious failures and safety violation scenarios and explain how the failures are detected. The failure model of Fides permits a server to misbehave but captures enough details in the transaction log for an auditor to detect the malicious failures as well as the failing servers.

Scenario 1: Incorrect Reads

A malicious server can respond with incorrect values for the data items read in the read requests. We use Lemma 1 to detect this.

Figure 4.10 gives an example of incorrect reads. Assume that the severs store bank details and there are two transactions T_1 and T_2 deducting \$100 from two accounts, x

and y. Block-10 contains T_1 and Block-11 contains T_2. T_1 reads two data items: one with id x, value 1000, r_{ts} = *ts-92*, and w_{ts} = *ts-88*, and the second with id y, value 500, r_{ts} = *ts-48*, and w_{ts} = *ts-48*. T_1 updates x to \$900 and y to \$400, and upon commitment, it also updates their r_{ts} and w_{ts} to *ts-100*. Any transaction executing after this must reflect the latest data. But T_2, committing at timestamp *ts-115*, has incorrect value of \$1000 for x (but up-to-date timestamps). This indicates that the server storing data items x is misbehaving by sending incorrect read values.

Scenario 2: Incorrect Block Creation

While executing TFCommit to terminate a transaction T_i, a malicious coordinator can add an incorrect Merkle Hash Tree (MHT) root of a benign server S_b in the block; this can cause audit failure of S_b (as Lemma 2 uses MHT roots to detect datastore corruption). But such an attempt will be detected by the benign server, as proved in Lemma 5.

In the *vote* phase of TFCommit, explained in §4.5.3, server S_b sends the MHT root corresponding to transaction T_i to the coordinator. If the coordinator stores an incorrect MHT root or a correct root but corresponding to an older transaction T_{i-1}, S_b can detect this in the *schResponse* phase of TFCommit. and not cooperate to produce a valid co-sign.

Scenario 3: Data corruption

A server may corrupt the data stored in the datastore, essentially not reflecting the expected changes requested by the clients. We assume a multi-versioned datastore in this example and use Verification Objects $\mathcal{VO}$ and MHT roots to detect datastore corruption, as proved in Lemma 2.

Consider a transaction T_i committed at timestamp *ts-100* and updated a data item x stored in S_m. Figure 4.11 indicates the data stored in server S_m that is being audited

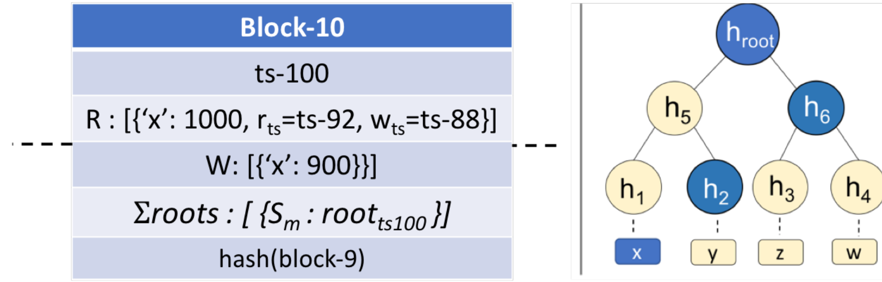

Figure 4.11: Data corruption example

at version *ts-100*. The auditor fetches the corresponding block (block 10) from the log and extracts x's value written by T_i and the MHT root corresponding to S_m. This MHT root should reflect x's updated value.

Assume S_m was malicious and did not update x to 900. In the next step of verification, auditor asks S_m for the $\mathcal{VO}$ of data item x at timestamp *ts-100*. S_m responds with $\{h_2, h_6, h_{root}\}$ (hash values of the sibling nodes of data x in the path from leaf to root). Auditor hashes x's value stored in the block ($H(900)$) and uses h_2 sent in $\mathcal{VO}$ to compute h'_5 and further, hash h'_5 and h_6 (from $\mathcal{VO}$) to compute the expected root, h'_{root}. This computed root should match the root the root stored in the block i.e., $h'_{root} = root_{S_m - ts100}$. But since S_m did not update the value of x to 900, the root computed by the auditor will not not match the root stored in the block (assuming collision-free hash functions). Thus data corruption at S_m, precisely at version *ts-100* is detected.

4.7 Evaluation

In this section, we discuss the experimental evaluation of TFCommit. Our goal is to measure the overhead incurred in executing an atomic commit protocol on untrusted infrastructure. The focus of Fides and TFCommit is *fault detection* in a non-replicated system, hence solutions based on replication that typically use PBFT [35] are orthogonal

to TFCommit.

In evaluating TFCommit, we measure the performance using two aspects: *commit latency* - time taken to terminate a transaction once the client sends end transaction request, and *throughput* - the number of transactions committed per second; TFCommit was implemented in Python. We deployed multiple database servers on a single Amazon AWS datacenter (US-West-2 region) where each server was an EC2 m5.xlarge vm consisting of 4 vCPUs, 16 GiB RAM and upto 10 Gbps network bandwidth. Unless otherwise specified in the experiment, each database server stores a single shard (or partition) of data consisting of 10000 data items.

To evaluate the protocol, we used Transactional-YCSB-like benchmark [44] consisting of transactions with read-write operations. Each transaction consisted of 5 operations on different data items thus generating a multi-record workload. The data items were picked at random from a pool of all the data partitions combined, resulting in distributed transactions. Although we presented TFCommit and Fides with the simplifying assumption of one transaction per block, in the experiments, we typically stored 100 non-conflicting transactions in each block. Every experimental run consisted of 1000 client requests and each data point plotted in this section is an average of 3 runs.

4.7.1 TFCommit vs. 2PC

As a first step, we compare the trust-free protocol TFCommit with its trusted counterpart Two Phase Commit [87]. TFCommit is essentially 2PC combined with the cryptographic primitives (Co-Signing and Merkle Hash Trees) which results in an additional phase due to the trust-free nature. Thus, comparing TFCommit with 2PC highlights the overhead incurred by TFCommit to operate in an untrusted setting. Both 2PC and TFCommit are implemented such that transactions are terminated and blocks are produced

sequentially so that the log does not have forks.

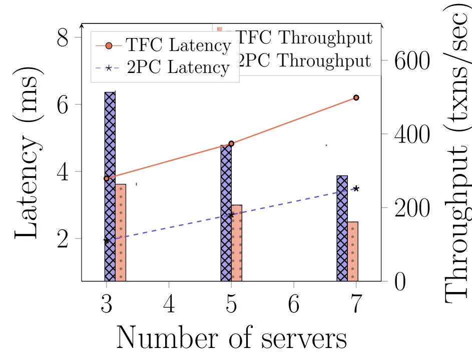

Figure 4.12: 2PC vs. TFCommit (TFC).

Figure 4.12 contrasts the performance of 2PC vs. TFCommit. We increase the number of servers and measure commit latency and throughput. In this experiment, each block stores *a single* transaction so that we can measure the overhead induced by TFCommit *per transaction*. Given that each block contains a single transaction and that blocks are generated sequentially, the servers are essentially committing one transaction after another.

As indicated in the figure, the average latency to commit a single transaction in an untrusted setting is approximately 1.8x more than a trusted environment. The throughput for 2PC is approximately 2.1x higher than TFCommit. TFCommit performs additional computations compared with 2PC: Merkle Hash Tree (MHT) updates to compute new roots after each transaction, collective signature on each block, and an additional phase. In spite of the additional computing and achieving trust-free atomic commitment, TFCommit is only 1.8x slower than 2PC. Having shown the overhead of TFCommit as compared to 2PC, the following experiments measure the performance of TFCommit by varying different parameters.

4.7.2 Number of transactions per block

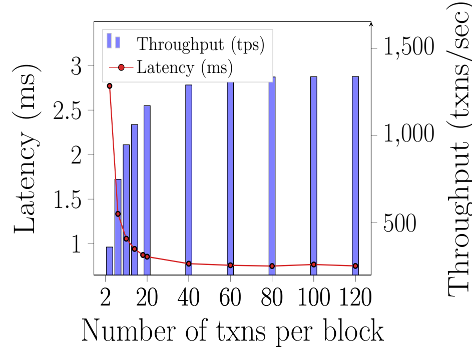

Figure 4.13: Varying number of transaction per block

In this experiment, we fix the number of servers to 5 and increase the load on the system by increasing the number of transactions stored within each block. Each database server consisted of 10000 data items. Figure 4.13 indicates the average latency to commit a single transaction and the throughput while increasing number of transactions stored within each block from 2 to 120. The latency to commit a single transaction reduces by 2.6x and the throughput increases by 2.5x when 80 or more transactions are batched in a single block. This experiment highlights that even though the blocks are produced sequentially, the performance of TFCommit can be significantly enhanced by processing multiple transactions in one block.

4.7.3 Number of shards

In this experiment, we measure the scalability of TFCommit by increasing the number of database servers (each storing a shard of 10000 data items) from 3 to 9, while keeping the number of transaction per block constant (100 per block). Figure 4.14 depicts the experimental results. The throughput of TFCommit increases by 47% and the commit latency reduces by 33% when the number of servers are increased from 3 to

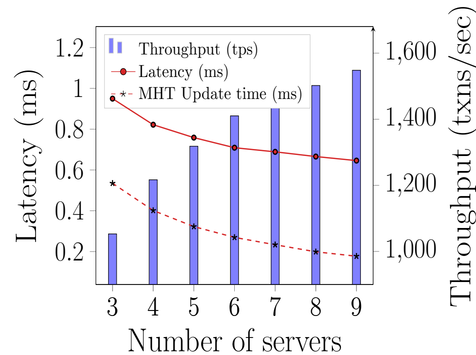

Figure 4.14: Varying number of servers.

9. Figure 4.14 also shows the most expensive operation in committing transactions i.e., Merkle Hash Tree (MHT) updates. Recall from §4.5.3 that in TFCommit, termination of each transaction requires computing the updated MHT root. Given that each block has 100 transactions, which in turn consists of 5 operations each, there are 500 operations in each block. With only 3 servers, all the operations access the three shards whereas with 9 servers, the 500 operations are spread across nine shards. Thus, the load per server reduces when there are more servers, resulting in the reduction of MHT update latencies. This experiment highlights that TFCommit is scalable and performs well with increasing number of database servers.

4.7.4 Number of data items

In the final set of experiments, we measure the performance of TFCommit by varying the number of data items stored in each database server, while keeping a constant of 100 transactions per block and using 5 database servers. The number of items stored in each server increased from 1000 to 10000 to measure the commit latency and throughput of TFCommit, as shown in Figure 4.15. The commit latency increases by 15% and the throughput reduces by 14% with the increase in number of data items per shard. The

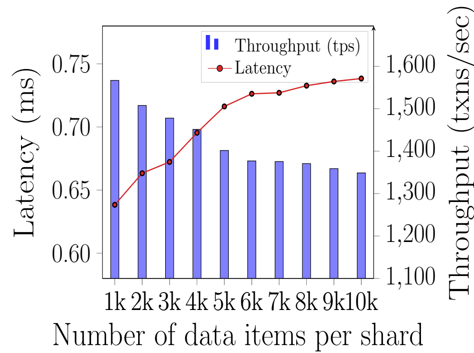

Figure 4.15: Varying number of data items per shard

performance fluctuation is due to the Merkle Hash Tree updates that varies with the number of data items. Updating a single leaf node in a binary hash tree with 1000 leaf nodes (data items) updates 10 nodes (from leaf to the root) and a tree with 10000 leaf nodes updates roughly 14 nodes. Thus, the performance of TFCommit decreases with increasing number of data items stored within each server.

4.8 Related Work

The literature on databases that tolerate malicious failures is extensive [68, 71, 208, 67, 140, 177]. All of these solutions differ from *Fides* as they: assume a singe non-partitioned database, rely on *replicating* the database to tolerate byzantine failures, and some also require a trusted component for correctness. Garcia-Molina et al.[68] were the earliest to propose a set of database schemes that tolerate malicious faults. The work presents the theoretical foundations on replicating the database on enough servers to handle malicious faults but lacks a practical implementation. Gashi et al. [71] discuss fault-tolerance other than just crash failures and provide a report composed of database failures caused by software bugs. HRDB by Vandiver et al. [208] propose a replication scheme to handle byzantine faults wherein a trusted coordinator delegates transactions

to the replicas. The coordinator also orders the transactions and decides when to safely commit a transaction. Byzantium by Garcia et al. [67] provides an efficient replicated middleware between the client and the database to tolerate byzantine faults. It differs from previous solutions by allowing concurrent transactions and by not requiring a trusted component to coordinate the replicas.

The advent of blockchains brought with it a set of technologies that manage data in untrusted environments. In both the open perimissionless and closed permissioned blockchains, due to lack of trust, the underlying protocols must be designed to tolerate any type of malicious behavior. But these protocols and their applications are mostly limited to crypto-currencies and cannot be easily extended for large scale distributed data management. Although permissionless blockchain solutions such as Elastico [141] Omniledger [112], and RapidChain [224] discuss sharding, it is with respect to transactions, i.e., different servers execute different transactions to enhance performance but all of them maintain copies of same data, essentially acting as replicas of a single database. These solutions differ from Fides as they focus of replicated data rather than distributed data.

In the space of transaction commitment, proposals such as [158, 230, 19, 226] tolerate malicious faults. Mohan et al. [158] integrated 2PC with byzantine fault-tolerance to make 2PC non-blocking and to prevent the coordinator from sending conflicting decisions. Zhao et al.[230] propose a commit protocol that tolerates byzantine faults at the coordinator by replicating it on enough servers to run a byzantine agreement protocol to agree on the transaction decision. Chainspace [19] proposes a commit protocol in a blockchain setting wherein each shard is replicated on multiple servers to allow executing byzantine agreement per shard to agree on the transaction decision. All these solutions require replication and execute byzantine agreement on the replicas, and hence differ from TFCommit. TFCommit uses Collective Signing (CoSi) [196], a cryptographic

multisignature scheme to tolerate malicious failures during commitment. CoSi has been adapted to make consensus more efficient in blockchains, e.g., ByzCoin [111]. To our knowledge, TFCommit is the first to merge CoSi with atomic commitment.

Fides uses a tamper-proof log to audit the system and detect any failures across database servers; this technique has been studied for decades in distributed systems [221, 220, 219, 96]. In [221] and [220], Yumerefendi et al. highlight the use of accountability – a mechanism to detect and expose misbehaving servers– as a general distributed systems design. They implement CATS [219] an accountable network storage system that uses secure message logs to detect and expose misbehaving nodes. PeerReview [96] generalizes this idea by building a practical accountable system that uses tamper-evident logs to detect and irrefutably identify the faulty nodes. More recent solutions such as BlockchainDB [99], BigchainDB [151], Veritas [2] and [66] use blockchain as a tamper-proof log to store transactions across fully or partially replicated databases. CloudBFT [170], on the other hand, tolerates malicious faults in the cloud by relying on tamper-proof hardware to order the requests in a trusted way.

The datastore authentication technique that uses Merkle Hash Trees (MHT) and Verification Objects was first proposed by Merkle [152]. The technique employed in Fides that enables verifing the datastore per transaction is inspired by the work of Jain et al. [106]. Their solution assumes a single outsourced database, and more importantly, it requires a central trusted site to store the MHT roots of the outsourced data and the transaction history. Fides replaces the trusted entity by a globally replicated log that stores the necessary information for authentication. Many works have looked at query correctness, freshness, and data provenance for static data but only few solutions such as [131] and [164] (apart from [106] discussed above) consider data updates. [131] and [164] discuss alternate data authentication techniques but also assume a single outsourced database.

4.9 Conclusion

Traditional data management systems typically consider crash failures only. With the increasing usage of the cloud, crowdsourcing, and the rise of blockchain, the need to store data on untrusted servers has risen. The typical approach for achieving fault-tolerance, in general, uses replication. However, given the strict bounds on consensus in malicious settings, alternative approaches need to be explored. In this chapter, we propose Fides, an auditable data management system designed for infrastructures that are *not* trusted. Instead of using replication for fault-tolerance, Fides uses fault-detection to discourage malicious behavior. An integral component of any distributed data management system is the commit protocol. We propose TFCommit, a novel distributed atomic commitment protocol that executes transactions on untrusted servers. Since every server in Fides is untrusted, Fides replaces traditional transaction logs with a tamper-proof log similar to blockchain. The tamper-proof log stores all the necessary information required to audit the system and detect any failures. We discuss each component of Fides i.e., the different layers of a typical DBMS comprising of a transaction execution layer, a transaction commitment layer, and a datastore. For each layer, both correct execution and failure detection techniques are discussed. To highlight the practicality of TFCommit, we implement and evaluate TFCommit. The experiments emphasize the performance and scalability aspects of TFCommit.

Chapter 5

QuORAM: A Quorum-Replicated Fault Tolerant ORAM Datastore

5.1 Overview

Privacy and security challenges due to the outsourcing of data storage and processing to third-party cloud providers are well known. With regard to data privacy, Oblivious RAM (ORAM) schemes provide strong privacy guarantees by not only hiding the contents of the data (by encryption) but also obfuscating the access patterns of the outsourced data. But most existing ORAM datastores are not fault tolerant in that if the external storage server (which stores encrypted data) or the trusted proxy (which stores the encryption key and other metadata) crashes, an application loses all of its data. To achieve fault-tolerance, we propose ***QuORAM***, the first ORAM datastore to replicate data with a quorum-based replication protocol. QuORAM's contributions are three-fold: (i) it obfuscates access patterns to provide obliviousness guarantees, (ii) it replicates data using a novel lock-free and decentralized replication protocol to achieve fault-tolerance, and (iii) it guarantees linearizable semantics. Experimentally evaluating QuORAM high-

lights counter-intuitive results: QuORAM incurs negligible cost to achieve obliviousness when compared to an insecure fault-tolerant replicated system; QuORAM **performs 1.4x better** in terms of peak throughput than its non-replicated baseline; and QuORAM performs **33.2x** better than an ORAM datastore that relies on CockroachDB, an open-source geo-replicated database, for fault tolerance.

5.2 Introduction

Due to the cloud's core policy of *pay-by-use*, individuals and organizations are increasingly shifting from managing their own storage servers to renting storage from third party cloud providers. Today, many products with high traffic, such as Twitter [206], Spotify [191], and Netflix [167], rely on cloud storage for some or all of their data storage requirements.

The cloud's convenience, however, comes at the cost of potentially compromising the privacy of the outsourced data. This privacy concern slows down the adoption of cloud services for many businesses [43]. Even with the data encrypted, users' access patterns can leak sensitive information to the cloud provider. Consider an example where a doctor stores patient records in a third-party cloud. If the doctor accesses a given patient's record more frequently than usual over a period of time, an intruder can infer some information about the patient's medical status. In fact, many works [105, 91, 110, 114, 34, 55] have shown concrete inference attacks by exploiting access patterns alone.

The privacy of outsourced data requires first to hide the data content through encryption, and then to *obfuscate* the access pattern to that encrypted data. Oblivious RAM, or ORAM, a cryptographic primitive originally introduced by Goldreich and Ostrovsky [78], achieves access pattern obliviousness. Although ORAM originally protected software executing on a single machine from an adversary on that same machine [78], ORAM's

functionalities are now extended to protect data accesses on remote storage [194, 192, 193, 23, 184, 136, 47, 36]. Summarizing the general idea in these works: they break up the data into logical blocks, each stored at a unique physical addresses on the external server. After each access to a logical block, the ORAM scheme shuffles the physical address, thereby mapping any sequence of logical memory accesses to a sequence of random physical memory accesses.

Broadly speaking, many remote ORAM system architectures [184, 47, 193, 23, 50] consist of three-layers: *an untrusted cloud storage server*, *a trusted proxy*, and *the clients*. An application encrypts its data under a key K and outsources the encrypted data onto an untrusted storage server. The trusted proxy holds the key K and accesses the storage server on behalf of the application's clients. Clients send read and write requests to the proxy, which then communicates with the server according to an *ORAM scheme* and responds back to the clients. An ORAM scheme translates client requests into a sequence of storage server accesses that are *indistinguishable* from other client request translations.

Recent proposals enhance the efficiency of ORAM schemes [184, 193, 47, 23, 216, 50, 36, 204, 27] by supporting concurrent and asynchronous client accesses. However, in most of these proposals, the proxy and the storage server are not fault-tolerant, deeming both components as single points of failure. If either crashes, the data becomes unavailable to users. Putting it differently, mitigating the privacy concerns of cloud storage derails one of the most significant advantages of the cloud: *fault tolerance*.

To date, Obladi [47] is the only ORAM system to tolerate crash failures without losing the system's state. For the storage server, Obladi relies on the standard fault-tolerance guarantees of cloud storage servers and assumes a highly available server. For the proxy, Obladi meticulously pushes 'valid' proxy states to the cloud storage such that after a crash, the proxy resets to the last valid state stored fault-tolerantly in the cloud. The main problem with this approach is that although a proxy's relevant state can be

recovered from the storage after a crash, the system cannot progress *while* the proxy is down. Moreover, delegating fault-tolerance to the cloud incurs higher latencies than an ORAM system with inherent fault-tolerance guarantees, as shown in the later sections of this chapter.

In distributed systems, the gold standard for fault tolerance is state machine *replication*. Zakhary et al. [223] discuss replication to tolerate failures in ORAM systems and demonstrate the challenges of employing standard design choices – such as locking and quorum based read-writes – in an ORAM system. The authors discuss only the risks of standard design choices for replication in ORAM systems rather than provide any solution to tolerate failures.

In this chapter, we present, ***QuORAM***, the first (quorum) replicated fault-tolerant ORAM system, consisting of multiple untrusted cloud storage instances and trusted proxies. QuORAM replicates the data on multiple storage instances, where each storage instance is accessed through its independent trusted proxy. A subset of these replicas serve each client request, thus allowing the system to tolerate some failures at both the storage and the proxy layers.

Serving client requests from only a subset of replicas raises the challenge of *consistency*, which we define using linearizable semantics: "each operation applied by concurrent processes [appears to take] effect instantaneously at some point between its invocation and its response" [98]. Note that the operations themselves need not take effect instantaneously across all replicas (and cannot, in the presence of asynchronous network delay); they only need to *appear* instantaneous to the clients. We address this challenge and prove that QuORAM guarantees linearizable semantics.

Apart from obliviousness and fault tolerance, QuORAM achieves the following additional functionalities:

1. It supports multiple concurrent reads and writes,

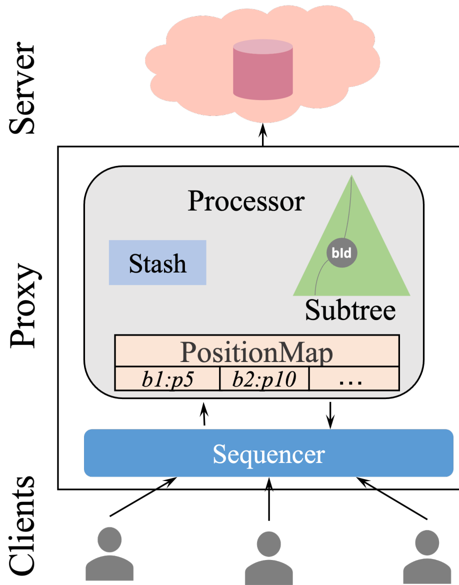

Figure 5.1: TaORAM's architecture

2. It has no single point of failure,
3. It replicates data across multiple (possibly colluding) cloud storage servers, and
4. It guarantees linearizable semantics.

In the rest of the chapter, §5.3 provides background on the ORAM scheme on which we build QuORAM; §5.4 describes the system and failure model of QuORAM; §5.5 defines security model of QuORAM; §5.6 proposes the replication and ORAM scheme designs on QuORAM; and §5.7 experimentally evaluates QuORAM with three baselines. §5.8 details security and linearizability proofs of QuORAM. §5.10 analyzes QuORAM's stash size and proves its space complexity to be *O(logN)*. §5.11 discusses related work and §5.12 concludes this chapter.

5.3 Background

This section introduces an ORAM scheme, TaORAM [184], that acts as a building block of QuORAM. TaORAM ensures obliviousness in the presence of concurrent, arbitrarily-scheduled accesses while preserving linearizable semantics. TaoStore's [184] ORAM scheme, TaORAM, builds upon another ORAM scheme Path ORAM [195]. Path ORAM organises data into a tree of *buckets*, each of which contains multiple data *blocks*. Path ORAM maps each block's position to a leaf node *lf*, and stores the block in any one of the buckets along the path from the root to that leaf *lf*. TaoStore [184] extends Path ORAM for asynchronous and concurrent queries. TaoStore's system architecture (Figure 1) consists of a storage server, a proxy, and the clients. The storage server stores the encrypted data in a tree and the clients access the data by sending read/write requests to the trusted proxy; the proxy accesses the storage server on behalf of the clients (using the encryption key it stores) according to the TaORAM protocol.

The proxy consists of two components: a *Sequencer* and *Processor*. The Sequencer communicates with clients and the Processor communicates with the server. The Sequencer maintains a FIFO request queue, which stores client requests in the order they arrive. When the proxy finds a response to a client request (after communicating with the server), the Sequencer forwards responses to clients in the request queue's FIFO order. The Processor maintains three pieces of local state: a position map, a local subtree, and a stash. The position map stores a block's leaf node id *lf* on whose path the block resides. The local subtree consists of blocks already fetched from the storage server (and possibly updated) but not yet written back, whereas blocks that do not fit in the subtree are stored in the stash. After the Processor fetches k paths, where k is a system configuration constant, a background thread writes those paths back to the server and deletes their contents from the local subtree. As k increases, the amount of memory consumed

by the proxy also increases.

At a high level, TaORAM executes the following steps for both reads and writes to a block B:

1) Let P be the path containing block B. TaORAM fetches P from the server if not already fetched; otherwise, it performs a *fake read* by fetching a random path.

2) TaORAM adds the read path to the local subtree. For write operations, it updates the value of B' in the local subtree.

3) TaORAM answers the client's request with B's value.

4) It assigns B to a new random path P' and updates the position map accordingly.

5) TaORAM next executes *flushing*: it reassigns each block in the subtree's path P or in the stash to the lowest non-full bucket intersecting with P and P', the block's newly assigned path. If no such bucket exists, TaORAM moves the block to the stash. TaORAM [184] proves that the stash size is bounded.

6) If TaORAM fetched k paths since the last write-back (where k is a system configuration constant), it writes the k paths from the subtree to the storage server. It then deletes all blocks in these k paths with no in-progress requests and retains blocks modified since initiating the write-back.

Although TaORAM preserves linearizability (as the authors proved in [184]), by itself, TaORAM does not tolerate failures. A user loses access to the data if the proxy or the storage server become unavailable. Additionally, the data cannot be recovered if the proxy or/and the storage server lose data.

5.4 System and Failure Model

Given the lack of fault-tolerance in TaORAM and almost all existing ORAM datastores, we propose QuORAM, an ORAM datastore that provides fault-tolerance via

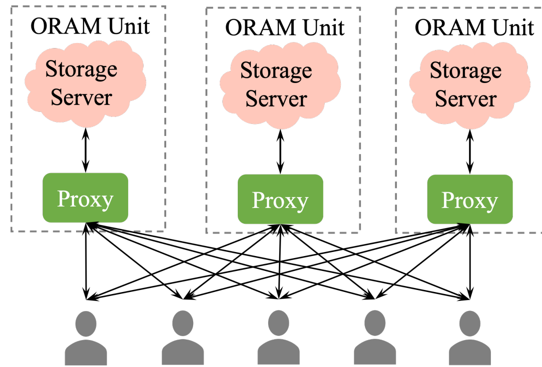

Figure 5.2: QuORAM Architecture

replication. This section presents the system and failure models of QuORAM.

5.4.1 System Model

QuORAM is a replicated oblivious data storage system that supports single key read and write operations on a key-value store, modeled as GET() and PUT() requests.[1] QuORAM has the same three-layered structure as a non-replicated ORAM system: untrusted storage servers to store encrypted data, trusted proxies controlled by the application to answer client requests by accessing the storage server, and clients who send read/write requests to the proxies. Typically in non-replicated ORAM systems, the overall state of the data is split between the proxy and the external storage. Extending an ORAM system to include replication also needs to maintain this one-to-one correspondence between a proxy and a storage server. Hence QuORAM replicates storage servers and proxies *in pairs* such that each proxy contacts exactly one storage server, and no two proxies contact the same storage server. We refer to a pair of ORAM server and proxy as an *ORAM unit* and depict the system architecture in Figure 5.2. Although not a requirement, since QuORAM aims to tolerate crash failures, we envision QuORAM to

[1] Inserts and deletes are modeled using GET() and PUT() requests.

be a geo-replicated datastore wherein the ORAM units and the clients accessing the data are all geo-distributed.

Within each ORAM unit, the external server S stores encrypted data while the corresponding proxy stores the respective secret key that encrypts S's data. The proxies in QuORAM also store other metadata necessary for the ORAM scheme (explained more in §5.6). All proxies in the system run the same ORAM scheme translating each ORAM operation into a sequence of storage server operations. From a client's perspective, it treats an ORAM unit as a black box that exposes a read-write interface.

5.4.2 Failure Model

Crash failures: Our goal in developing a replicated ORAM system is to provide durability and failure tolerance comparable to production cloud storage. An ORAM unit enters a failed state when its storage server and/or its proxy crashes or when network partitions occur. These failures are effectively equivalent to the *entire* unit being unreachable: since the proxy holds the encryption secret key, the data accessed from the storage server cannot be decrypted without the proxy's decryption key, and the proxy's key is useless without the data from its corresponding storage server. As such, we consider an ORAM unit failure to be a single failure event, regardless of which component actually failed.

To tolerate a maximum of f failures, QuORAM replicates data onto $2f+1$ ORAM units. When a failed unit (server and proxy) resumes operation after a crash, it resumes the state before the crash. If an application assumes that a failed unit does *not* recover its previous state upon crash recovery, then the recovered unit can copy the current state from a *majority* of the ORAM units (this is because QuORAM uses majority quorums to replicate the data and reading data from a majority guarantees reading the latest values of data, as will be discussed in §5.6.1).

All communication channels – clients to proxies, proxies to servers – are asynchronous, unreliable, and insecure. All communication channels are made secure using encryption mechanisms such as transport layer security or secure socket layer to mitigate message tampering.

Threat model: QuORAM assumes an *honest-but-curious* adversary that executes the designated protocol correctly. An adversary may control one or all external storage servers and can observe, track, and analyze data accesses to and from the server and perform inference attacks based on the access patterns. The adversary can control the *asynchronicity* of the network and also schedule read/write requests via a compromised client. Crash failures are consistent with the *honest-but-curious* adversarial model, hence we do not consider more severe malicious failure modes in this chapter. The goal is to design an oblivious data storage system that tolerates catastrophic crash failures under the aforementioned adversarial model.

5.5 Security Model: Obliviousness in a Replicated ORAM Setting

Existing definitions of obliviousness are insufficient to capture the security of a replicated ORAM system because even if a single proxy-server pair provides ORAM guarantees, the choice of replication protocol may leak non-trivial information. Consider quorum based replication protocols such as CRAQ [200] or Hermes [108]. In these works, read requests access a single node (i.e., single-node read quorums) and write requests access *all* the nodes in the system (i.e., all-node write quorums, which intersect with all single-node read quorums). Deploying such schemes allows an adversary to distinguish between read and write operations by merely observing how many units are accessed for

an operation, regardless of whether the ORAM scheme leaks any information about the operation type.

To formalize the above information leak, we develop a new definition of obliviousness, adapted from the notion of *aaob-security* (adaptive asynchronous obliviousness) from TaORAM [184]. Intuitively, an ORAM scheme is aaob-secure if any two sequences of operations and any two data sets are indistinguishable to the attacker. This section first defines the ORAM scheme of QuORAM and then presents a security game based on which we define the security of replicated ORAM datastores.

5.5.1 ORAM scheme definition

A typical asynchronous ORAM scheme consists of two modules ORAM = {Encode, OClient}. Encode encrypts data D, and produces D_{enc} and a secret key K. An external server stores D_{enc} and a stateful ORAM client, OClient, stores K. QuORAM uses the above definition of ORAM = {Encode, OClient} for individual ORAM units but extends it to a list: Rep-ORAM = (ORAM$_1$, ORAM$_2$, ..., ORAM$_n$) for n ORAM units. Each ORAM unit ORAM$_i$'s Encode module receives the same data D. Given D, the Encode module outputs a secret key K_i and the data set D_{encK_i} encrypted using K_i after internally shuffling the data in a random order. The shuffling mitigates identical access patterns across different storage servers at the beginning of execution. The ith external server stores D_{encK_i} and the corresponding i^{th} OClient retains K_i – both the server and OClient (executed by proxy) form an ORAM unit, ORAM$_i$.

Individual OClient's execute ORAM requests denoted as (op, bid, v) where op $\in$ {read, write}, bid represents a data block's id, and v=$\perp$ for reads or a new block value for writes. These operations result in read/write accesses to the storage server. While an OClient process recognizes a single type of operation – ORAM operation – represented by (op, bid,

v), QuORAM distinguishes between two types of operations: *logical* and *ORAM*. Logical operations are client requested read/write operations[2] represented as (lop, bid, v) – where lop $\in$ {read, write}, bid is a data block's id, and v=$\perp$ for reads or an updated value for writes. Each logical operation in-turn translates to a sequence of ORAM operations (op, bid, v)$_i$ where i identifies an ORAM unit. For example: a logical read can translate to a set of ORAM reads sent to a quorum of ORAM units followed by ORAM writes sent to that quorum.

5.5.2 Security definition

A replicated ORAM system, such as QuORAM, requires a slightly different security definition compared to aaob-security. The attack presented at the beginning of this section of using CRAQ [200] or Hermes [108] replication protocol clearly indicates that an aaob-secure system can still leak the type of logical operation. Hence, we extend aaob-security to include *logical obliviousness* i.e., *l-aaob-security*. *l-aaob-security* is an indistinguishability based security definition, which we define using a game $\mathcal{G}$. The steps of the game are:

- The game picks a uniformly random bit $b \in \{0, 1\}$, called the challenge bit.
- An adversary $\mathcal{A}$ generates two same-sized sets of data D_0 and D_1. The game calls Rep-ORAM on D_b, i.e., it calls $D^b_{encK_i}, K_i \leftarrow \mathsf{Encode}_i(D_b)$ for each ORAM unit i. The external server and OClient of an ORAM unit i store the encrypted data D_{encK_i} and the secret key K_i, respectively.
- The adversary, at any point in time, schedules two logical operations $(lop_{i,0}, lop_{i,1})$ consisting of arbitrary logical reads/writes. The game picks only one of the operations $lop_{i,b}$ and executes a replication protocol chosen by the replicated ORAM

[2] Logical reads/writes are equivalent to a key value store's GETs/PUTs.

system by sending ORAM read/write operations to the ORAM units. The game notifies the adversary once the operation terminates without revealing the actual result, as the adversary can easily guess the challenge bit b based on the result.

- Throughout the above process, the adversary can read, delay, drop, and learn the timing of (but not modify) messages. The adversary can also cause any storage server, proxy, and/or client to crash, with at most f proxy/storage server failures.
- Finally, after scheduling any number of logical operations, the adversary decides on the value of the challenge bit b. The game $\mathcal{G}$ returns True if the adversary chooses the right bit; and otherwise returns False. At this point, the game terminates.

We define *l-aaob-advantage* of the adversary $\mathcal{A}$ against Rep-ORAM as

$$Adv^{l-aaob}_{Rep-ORAM} = 2 * Pr[\mathcal{G}^{l-aaob}_{Rep-ORAM} \Rightarrow \mathsf{True}] - 1 \quad (5.1)$$

A replicated ORAM system is *l-aaob-secure* if $Adv^{l-aaob}_{Rep-ORAM}$ is negligible for any polynomial time adversary $\mathcal{A}$, i.e., any polynomial-time adversary can guess the challenge bit with probability negligibly higher than half. In other words, an ORAM scheme is *l-aaob-secure* if any two sequences of logical operations[2] and any two data sets are indistinguishable to the attacker.

5.6 QuORAM: a replicated ORAM datastore

This section presents the design of the replicated ORAM datastore, QuORAM. In designing QuORAM, we aim to achieve three goals: (i) obfuscate the access patterns to achieve privacy and *l-aaob-security*, (ii) replicate the data for fault-tolerance, and (iii) achieve the above two goals while preserving linearizable semantics.

To describe how we achieve the above goals, this section first discusses the design of a data replication protocol that preserves linearizability, followed by the ORAM scheme that hides access patterns.

5.6.1 QuORAM's replication protocol

To describe QuORAM's replication protocol, for now, we assume the system employs a state-of-the-art ORAM algorithm, TaORAM, as a black-box (this is relaxed in §5.6.2) and focus only on the replication protocol that provides linearizability guarantees. Choosing an existing replication protocol or designing one is a non-trivial task due to preserving obliviousness. To highlight the challenges in replicating an ORAM datastore, we propose a naive solution followed by QuORAM's replication design.

Naive solution:

As discussed in §5.5, using optimized replication solutions such as Hermes [108] or CRAQ [200] breaks obliviousness because they access varying numbers of replicas for logical read and write operations. The naive solution presented here mitigates the above challenge by deploying a single round replication protocol wherein a client accesses the same number of ORAM units for both read and write operations. Note that to ensure linearizability, the sites that handle read and write requests, *read quorum* and *write quorum*, must intersect with each other (e.g., majority quorums). In this single round multicast protocol, assuming majority quorums, a client reads from a majority and writes to a majority of the ORAM units.

While this solution is efficient since a client communicates with the ORAM units only once, it violates linearizability. We show how this solution breaks linearizability by providing an example. Consider a system with 3 replicated ORAM units where clients read or write from 2 out of the 3 replicas. A client *cl1* sends a write request for a data item

identified by key k, ($k = v'$) to ORAM units 1 and 2. Since the communication channels are asynchronous, assume that ORAM unit 1 receives the request and updates k's value to v' while ORAM 2's write request is in-transit. Now, another client $cl2$ performs two consecutive reads on key k once from ORAM units 1 and 2 and subsequently from ORAM units 2 and 3. For each request, the client picks a read value corresponding to the latest timestamp (typically achieved using totally ordered timestamps [123]). For the first request, the client reads the most up-to-date value v', whereas for the second request, it reads only the older value of k.

This is a linearizability violation as from the external client's perspective, the operations on k appear non-linear.

To circumvent this problem, the proxies can either deploy a locking mechanism (as is typical in database systems and as is done in Hermes [108]) or add another round of communication to ensure correct ordering of requests. But employing a locking mechanism can breach obliviousness as locking leads to deadlocks and detecting/resolving deadlocks in distributed systems requires additional communication across replica units. Since the adversary controls all communication channels, such additional communication leaks non-trivial information. Due to these reasons, QuORAM replicates data using a lock-free approach that uses two rounds of communication between a client and the ORAM units.

QuORAM's replication

QuORAM's replication protocol design is inspired by Lynch and Shvartsman's replication protocol [142]. In designing the replication protocol, we follow the abstractions defined in the Consensus and Commitment (C&C) framework [148], which consists of four phases: *Leader election, Value Discovery, Fault-tolerance,* and *Decision.* The C&C framework [148] describes that most replication protocols are centralized in that one of

the replicas acts as a *leader* and drives the protocol by communicating with other replicas. In such compositions, the leader node can be overloaded and become a bottleneck.

QuORAM chooses a different, decentralized approach where a client interested in reading or writing the data takes on the role of a leader and communicates with all ORAM units. This choice reduces the additional overhead on a single leader unit and avoids an adversarial case where an adversary delays the leader's communication links, thwarting the system performance.

Following the abstractions of the C&C framework, QuORAM's replication has two phases: in the first phase, a client identifies the most up-to-date value of an item by reading from a read quorum and in the second phase, it writes either the identified value (for read requests) or the updated value (for write requests) onto a write quorum of ORAM units, where the read and write quorums have non-empty intersection. Using the terminology of Lynch and Shvartsman's protocol [142], we term the first phase as the *query* phase and the second as the *propagate* phase. Given that some replica units' states may diverge due to crash or network failures, to easily identify the most up-to-date value of a given data item, each data item in QuORAM additionally maintains a monotonically increasing *tag* consisting of a sequence number and client id, $t =< seqNum, clientId >$. This is analogous to version or timestamp based datastores.

Overview: Figure 5.3 represents a high-level description of QuORAM's replication protocol. A client that wants to logically read or write a key k executes the replication protocol in two phases: query and propagate. The client first sends ORAM read requests for key k to a read quorum of ORAM units and waits to receive a response consisting of value v and tag t from the read quorum. The actions of the propagate phase depend on the type of client request: for logical reads (GETs), the client selects the value v with the highest tag t and multicasts ORAM write with v and t to a write quorum of units. For logical writes (PUTs), the client creates a new tag t' by incrementing the highest tag t (how is

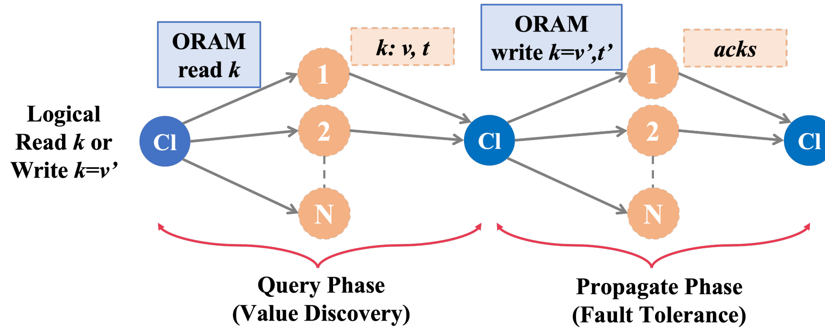

Figure 5.3: QuORAM's replication protocol. Each circle represents an ORAM unit a client *Cl* executes the protocol

explained later) and multicasts ORAM write with v' and t' to a write quorum of units where v' is the new value. Upon receiving the ORAM write request, proxies in QuORAM update the value and tag *if and only if* the received tag t' is greater than its own tag value. The propagate phase terminates when the client receives acknowledgments from the write quorum. *For both logical read and write requests, a client considers its request to be complete only after completing both phases.*

From this overview, it is clear that if a client uses different read and write quorums in the query and propagate phases, then both sets of quorum fetch a path, shuffle, and write it back onto external servers. This creates unnecessary bandwidth and compute overheads. QuORAM addresses this issue by using the same quorum for both query and propagate phases. Since QuORAM reuses read and write quorums interchangeably, we stop distinguishing between read and write quorums and impose a requirement that *any two quorums must intersect with each other* (rather than imposing read and write quorums must intersect). This way, a client can pick any quorum and use it in query and propagate phases. While for simplicity, QuORAM uses majority quorums [201], i.e., sets of $\lceil (N+1)/2 \rceil$ ORAM units, the application can pick any other quorum composition that guarantees non-empty intersection between any two quorums (e.g., tree quorums [1] or

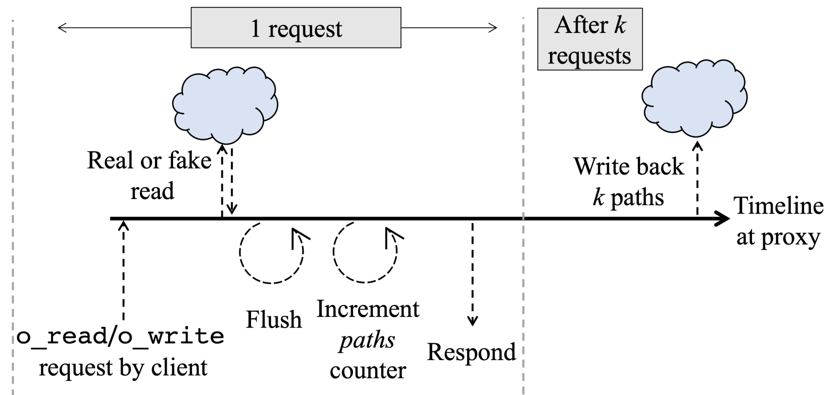

Figure 5.4: Timeline of the proxy in TaORAM

grid quorums [163]). Informally, using the same quorum for both the query and propagate phases does not leak any additional information since an attacker already observes what ORAM units are accessed while querying.

QuORAM's choice to communicate with only a quorum of ORAM units, instead of all, may result in a client not receiving a full quorum of responses (due to individual unit failures or message losses), even if globally, a majority of the units are alive. To ensure system progresses as long as a majority of ORAM units are live, we use timeouts to detect an unresponsive unit in a quorum and replace it with another. This brings us to the final design of QuORAM's replication protocol, whose pseudocode is described in Algorithm 7. Algorithm 7 and the rest of the chapter distinguishes logical reads and writes from ORAM reads and writes by denoting logical operations as `l_read` and `l_write` (indicating GET() and PUT() requests respectively of a key value store), and ORAM operations as `o_read` and `o_write` (representing the query and propagate phase messages respectively). Algorithm 7:

1. A client C that either wants to logically read or write a block bId starts the protocol by picking a quorum Q of randomly chosen majority of ORAM units (line 1).

Algorithm 7 Pseudocode for QuORAM executed by a client with id cId for an operation of $opType \in$ `l_read, l_write` on block bId and value v.

Query Phase:

1: Q ← randomly select a set of $\lceil (N+1)/2 \rceil$ ORAM units
2: opId ← a globally unique operation ID
3: Multicast `o_read(bId, opId)` to all ORAM units in Q. Collect each response (v_i, tag_i), where tag_i is a tuple of ($seqNum_i$, cId_i)
4: While waiting for all responses from Q, if a read request sent to ORAM unit U times out:
   (a) U′ ← randomly selected unit not in Q
   (b) Q ← Q + U′ - U
   (c) Send `o_read(opId,bId)` to U′
5: Upon receiving responses from all Q units, select the response r with the highest tag
6: If opType = `l_write`, set $t' \leftarrow (r.tag.seqNum + 1, cId)$ and $v' \leftarrow v$
7: If op_type = `l_read`, set $t' \leftarrow r.tag$ and $v' \leftarrow r.v$

Propagate Phase:

8: Multicast `o_write(opId,bId,`v'`,`t'`)` to all units in Q
9: While waiting for all responses from Q, if a write request sent to ORAM unit U times out:
   (a) Execute steps 4(a) to 4(c)
   (b) Send `o_write(opId, bId, `v'`, `t'`)` to U′, *without changing* t' *and* v' *sent in Step 8*
10: Upon receiving acknowledgements from Q, the client considers the (logical) operation complete

2. The client assigns its operation a globally unique operation id, $opId$, (e.g., a sequence number and a client's unique id) as shown in line 2. This $opId$, a separate identifier from a data item's tag, is important to identify in-progress operations at both the client and proxies.

3. The client then multicasts `o_read(bId, opId)` to the proxies in quorum Q, who in-turn may fetch the block and the associated tag from their respective storage servers and retain it in the subtree until the block is written back (see §5.6.2 for the steps executed by a proxy). The client waits to receive responses consisting of

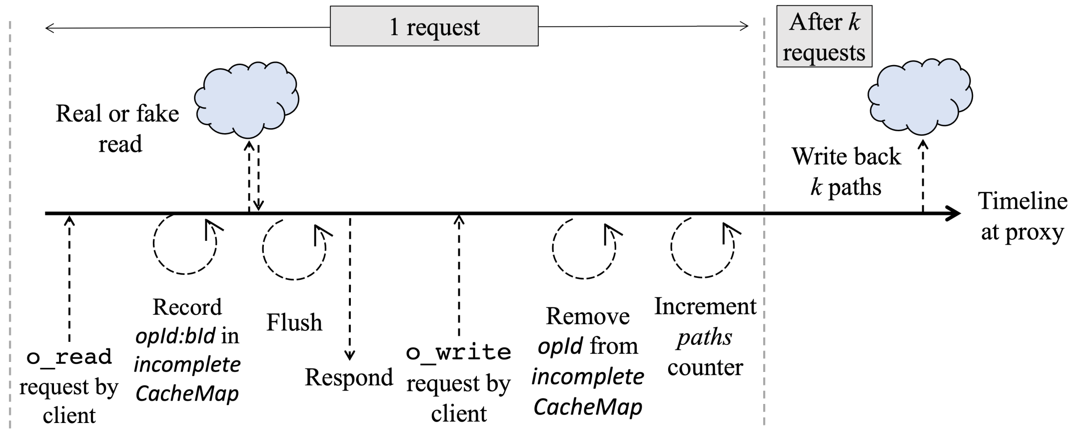

Figure 5.5: Timeline of a proxy in QuORAM. Figures 5.4 and 5.5 capture the difference between the functionalities of a proxy in TaORAM vs. a proxy in QuORAM.

the block's value *and* tag from *all* proxies in Q (line 3).

4. If the client times-out while waiting for a response from an ORAM unit U, it updates its quorum by removing U and adding another randomly selected unit U' to Q. The client then sends the `o_read` request to U'.

5. Upon receiving Q responses, the client picks the response r with the highest tag (line 5).

6. If client C's operation is `l_write`, it updates the tag (t') by incrementing the sequence number of the highest tag and updating the tag's client id to C's id and sets the value (v') to the block's new value v.

7. If client C's operation is `l_read`, it retains the highest tag (t') and its corresponding value (v') of the response r identified in Step 5.

8. Client C then broadcasts `o_write(opId, bId,` v'`,` t'`)` with the respectively updated value v' and tag t' to the proxies in Q and waits for their acknowledgements. A proxy P that receives the `o_write()` message sends an acknowledgement to C.

However, the proxy P updates the value and tag ***if and only if*** the received tag t' is greater than its own tag value.

9. If the client times-out while waiting for an acknowledgement from a unit U (line 9), the client re-executes steps 4(a) to 4(c), essentially updating the quorum Q and sending `o_read` to the newly added unit U′. The client then sends the `o_write` request to U′, *without changing the value v' or tag t' sent in Step 8*, which is important to preserve linearizability. Note that even though only the write part of the operation timed-out, the client sends `o_read` before retrying `o_write` on the newly added unit to ensure the proxy fetches the necessary block and update its data structures accordingly.

10. Once the client receives acknowledgements from the quorum Q, the client considers the logical operation to be successful.

This concludes the discussion of QuORAM's replication protocol. This protocol guarantees linearizability, which is discussed in §5.6.3.

5.6.2 QuORAM's ORAM Scheme

Having discussed the replication protocol of QuORAM that preserves linearizability, this section discusses QuORAM's goal of providing obliviousness by hiding access patterns. QuORAM builds its ORAM scheme on top of TaORAM, described in §5.3 and we suggest reviewing it before proceeding.

Challenge of using TaORAM as-is: If proxies in QuORAM implement the ORAM scheme as-is in TaORAM, for each logical request the proxies fetch the requested block's path twice and write it back to the server twice, incurring unnecessary communication and compute overhead. The reason for the inefficiency is as follows: based on the replication protocol described in §5.6.1, in a single execution of the protocol, a given proxy

is either part of the quorum or not. If part of the quorum, the proxy always receives an `o_read` request in the query phase followed by an `o_write` request in the propagate phase, irrespective of the type of logical request (Figure 5.3). Recall from §5.3 that for every ORAM request, TaORAM fetches a path, flushes it, and writes it back (after k requests) to the server. If the proxy treats them as two separate and independent ORAM operations, then it fetches a path (real or fake) and writes it back to the server for ***both*** ORAM requests, incurring unnecessary overhead.

Solution: To mitigate the double fetching/writing of a block's paths, all proxies in QuORAM treat the two ORAM operations as correlated, and execute a single fetch and a single write-back for each logical operation. We discuss what happens when an adversary suppresses an `o_read` or `o_write` later. Figures 5.4 and 5.5 illustrate the details of a proxy's interactions between a client and it's external storage in QuORAM and contrasts them with the corresponding interactions in TaORAM. We now discuss in more details how QuORAM manages the execution of logical operations.

Challenge of asynchronously receiving `o_read` and `o_write`: QuORAM considers an `o_read` followed by an `o_write` as a single client's request but they arrive sequentially; an adversary who controls the communication channels can control the interval between the two ORAM requests. This implies a proxy needs to *remember* for which request it has already fetched a path from the server and for which request it has not.

Solution: We achieve this by introducing a new data structure in TaORAM's Processor: incompleteCacheMap, as depicted in Figure 5.6. The incompleteCacheMap tracks client operations that are read but not written by mapping an operation to its requested block, i.e., $opId$ to bId. If multiple operations access the same block, the incompleteCacheMap tracks them all. For the incompleteCacheMap, we use an LRU-based cache with a bounded number of elements for our evaluations (but any other cache design can be used). The size of the incompleteCacheMap is a system configuration and we assume

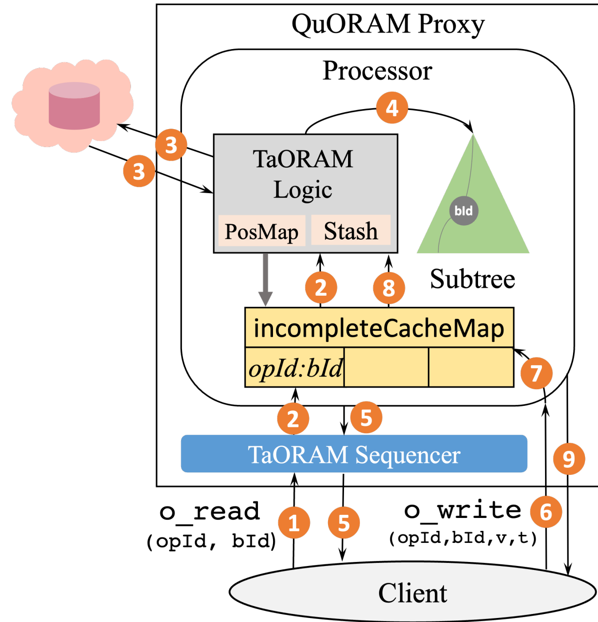

Figure 5.6: QuORAM's ORAM scheme built atop of TaORAM.

this size is *not* hidden from the adversary.

Another change in QuORAM's ORAM scheme compared with TaORAM is in deciding when to write-back fetched paths (Figures 5.4 and 5.5). Conceptually, both ORAM schemes write-back k paths to the server after serving k requests, where k is a system configuration and both schemes track the number of requests served with a counter denoted by $paths$. But the main difference lies in how the two schemes define a single client request: TaORAM considers an `o_read` or an `o_write` as an independent, single client request, whereas QuORAM considers an `o_read` followed by an `o_write` with matching $opId$ as a single client request. Due to this difference, TaORAM increments $paths$ immediately after fetching a path from the server, indicating the accessed path is ready to be written back; whereas QuORAM waits until receiving the corresponding `o_write` before incrementing $paths$. Both schemes write-back when the $paths$ counter value is a multiple of k.

Figure 5.6 provides the step-wise interactions between the various components of QuORAM. In the figure, Subtree, TaORAM Logic, and TaORAM Sequencer denote TaORAM's unmodified subtree, Processor and Sequencer logic (see Section 5.3). The steps depicted in Figure 5.6 are as follows:

1. A client sends an `o_read(opId,bId)` request to a quorum of proxies (Figure depicts interaction with one). The unmodified TaORAM Sequencer records the request and forwards it to the Processor.

2. The Processor adds a new entry $opId : bId$ to the incompleteCacheMap. If the cache is full, it evicts an entry based on the cache policy before adding the new entry; cache eviction increments $paths$ (§5.6.2 describes the reasoning). The Processor then forwards the request to the TaORAM Logic, which abstractly represents all the unmodified data structures and execution logic of TaORAM's Processor.

3. The TaORAM Logic then fetches a path - real or fake - from the external server.

4. The Processor moves the fetched path, real or fake, to the Subtree.

5. Irrespective of real or fake reads from the server, the Processor sends the read response back to the client, through the Sequencer. For fake reads, the block's real value can be found either in the Subtree or Stash. For real reads, the Processor assigns the block bId to a new path. The Processor then flushes the fetched path – real or fake (see §5.3 for details on flushing).

6. The client (after receiving responses from a quorum and updating the value and tag according to Algorithm 7) sends an `o_write(opId,bId,v,t)` to the chosen quorum of proxies.

7 Since `o_write` requests *do not* access the external server, they can be processed directly by the Processor bypassing the Sequencer, without breaking obliviousness. Upon receiving `o_write`, the Processor of a proxy checks if the incompleteCacheMap has entry for $opId$ and bId : if yes, it executes step 8; if no, i.e., the cache evicted $opId : bId$ entry in between `o_read` and `o_write`, then it executes step 9 by sending a *negative* acknowledgement to the client, indicating this request has failed.

8 The Processor removes $opId : bId$ entry from the incompleteCache, increments the $paths$ counter and forwards the `o_write` request to TaORAM Logic. When $paths$ reaches a multiple of k, TaORAM Logic asynchronously writes back k paths to the server. After receiving a write acknowledgement from the server, TaORAM Logic deletes the k paths from the Subtree. Importantly, while deleting the paths, TaORAM Logic *does not* delete blocks that are pointed to by the incompleteCacheMap.

9 The Processor then sends a positive acknowledgment to the client, and after receiving acknowledgments from the chosen quorum, the client considers its operation complete. If a client receives at least one negative acknowledgement from any proxy, it deems its request as unsuccessful. Based on the application, the client may retry the failed request.

Discussion on incompleteCacheMap eviction

Along with tracking ongoing client requests, incompleteCacheMap's other main role is to limit an adversary from causing a memory overflow at a proxy. An adversary can send only `o_read` messages of clients and suppress all `o_write` messages. Because the ORAM scheme fetches paths on `o_reads` and it writes-back paths and clears their memory upon receiving k `o_writes`, if a proxy receives only `o_reads` without any `o_writes`, its memory can overflow. To mitigate such adversarial behavior, we choose a limited size

cache-like datastructure that dictates how many in-progress requests a proxy can serve at a given time. As described in Step 2, if the Processor finds incompleteCacheMap to be full when a new `o_read` arrives, it evicts an entry based on the cache eviction policy and increments the *paths* counter. The counter increment is necessary to ensure a proxy writes-back paths *even if it receives no* `o_write`*s*. Because we assume an adversary knows the incompleteCacheMap size, writing k paths back after k combined `o_write`s and cache evictions does not leak any non-trivial information to an adversary.

An important detail for obliviousness and linearizability lies in the details of what happens when a block gets evicted from the incompleteCacheMap. *Eviction from incompleteCacheMap does not mean eviction from the proxy.* Eviction merely allows the proxy to *forget* that the evicted block had an in-progress request and allows the proxy to treat it as a block whose logical operations are complete. When the incompleteCacheMap evicts an entry, $opId : bId$, the operation's `o_write` request becomes a no-op because whatever the proxy read in the `o_read` operation is no longer guaranteed to be present in the proxy. Hence, the proxy notifies a client if its `o_write` request failed by sending a negative acknowledgement (7) and the application can decide how to handle negative acknowledgements. We assume that the adversary knows the incompleteCacheMap size; hence revealing the type of acknowledgement – positive or negative – to the adversary *does not* break obliviousness.

Discussion on a proxy's memory usage

As discussed earlier, QuORAM writes-back k paths to the server after serving k client requests. But as seen in step 8, after a write-back completes QuORAM deletes only those blocks with no pointers in the incompleteCacheMap (i.e., QuORAM retains blocks accessed by ongoing requests).

Memory Issue: QuORAM's logic of not deleting certain blocks in the k paths

after a write-back can cause a proxy's memory, i.e., Subtree, to grow unbounded if the retained blocks are never accessed again (a larger Subtree may indirectly cause a larger Stash). To see why, we consider a simple example where $k = 1$ and two concurrent logical operations $op1$ and $op2$ access the same block, $b1$. Say a proxy receives $op1$'s `o_read` first, upon which it fetches a real path containing $b1$ from the external server. While the path is being fetched, it receives $op2$'s `o_read` and since the proxy already asked to read $b1$'s real path, it reads a fake path from the server for $op2$. When both `o_read`s are answered, the proxy receives $op1$'s `o_write`, which increments *paths* and initiates a write-back (because $k = 1$). The proxy writes the path back but cannot delete $b1$ because it has not yet received $op2$'s `o_write` request (and $op2$ read a fake path). If $op2$ updates the block and the path that block $b1$ resides on is *never* accessed and hence never written back again, then $b1$ may permanently reside in the proxy. If many such contending requests occur for different blocks at k write-back boundaries, a proxy's memory may grow unbounded. We note that in practical scenarios, this type of memory growth is improbable since clients will likely access some block in $b1$'s path over time and $b1$ will be opportunistically written back to the server, freeing it's memory. But the unbounded memory issue is a theoretical possibility.

Solution: To mitigate the unbounded memory growth problem, QuORAM creates a daemon process in the proxies wherein the daemon process *simulates* a client access every preset interval of time (e.g., 100 ms). The background process mimics both `o_read` and `o_write` requests within a proxy and that proxy fetches a path – real or fake – in accordance with the ORAM algorithm, flushes the path, and writes-back k paths after k accesses, *including the accesses generated by the background process.* We assume the adversary is aware of this behavior where irrespective of client requests, each proxy performs its own access at regular intervals.

To further ensure that a proxy's Subtree (and hence it's Stash) does not grow in between the access intervals, we add a new datastructure called *excessBlocks*. Going back to the memory issue example, excessBlocks stores all blocks retained by the proxy after a write back to accommodate ongoing client requests. Introducing this new datastructure modifies Step 8 of the ORAM logic: after receiving a write acknowledgement of k paths from the server, a proxy moves all blocks in those k paths that are pointed to by the incompleteCacheMap and which would have otherwise been deleted by TaORAM to excessBlocks. This allows TaORAM Logic to free up all k paths from Subtree. In the evaluation section §5.7, we experimentally show that the size of excessBlocks remains low ($c \cdot logN$, where c is a constant), irrespective of contention in the workload. §5.10 formally analyzes the size of Stash, which is of order $O(logN)$, as well as the space utilization of a proxy.

Regarding how the daemon process selects blocks to access, it can be sequential, pseudorandom, or blocks in excessBlocks. If an application chooses to access blocks in excessBlocks, it must be noted that only blocks with no entries in incompleteCacheMap can be accessed and if no such blocks exist or if excessBlocks is empty, then the daemon process *must* continue to access blocks at preset intervals of time. Intuitively, how the daemon process selects blocks has no implications on obliviousness because this process *simulates* client requests; if an ORAM scheme hides how and what blocks are accessed by clients, then it also hides how and what blocks are accessed by the background process.

5.6.3 Security and linearizability of QuORAM

SECURITY:

The following theorem captures QuORAM's security.

Theorem 1: *Assuming individual ORAM units are aaob-secure, QuORAM is l-aaob-*

secure.

§5.8 describes the detailed proof of the theorem. The core idea of the proof lies in how QuORAM replicates data: for *all* types of logical requests, QuORAM executes a query phase followed by a propagate phase. Both phases access the same number (i.e., majority) of ORAM units, even in the presence of failures. All system configurations – k the write-back frequency parameter, the size of the incompleteCacheMap, and the access interval of a proxy's daemon process – are known to an adversary, and hence any decision made based on these configurations do not leak any new information to an adversary.

LINEARIZABILITY:

Theorem 2: *QuORAM provides linearizability.*

Arguing for linearizability – defined per data item – in replicated data systems, especially semi-honest ones, is non-trivial. §5.9 provides a detailed proof of how QuORAM guarantees linearizable semantics.

Intuitively, QuORAM's linearizability proof captures two main relations between any two operations: (i) the tag values of any two completed logical operations have a strict less-than or less-than-or-equal-to relation; and (ii) a given logical operation – read or write – is atomic. The former point captures the relative ordering of logical operations and this order is particularly important for conflicting operations. The latter point implies that if an operation op_i wrote a block, then an operation op_j immediately succeeding op_i must read the block written by op_i; and if operation op_i merely read a block without writing it, then operation op_j immediately succeeding op_i must also read the same value as op_i. We further note that even a compromised client executing QuORAM's replication protocol does not violate linearizability.

	N.California	Ohio	N. Virginia
N. California	6.3ms	51.32ms	62.19ms
Ohio	53.34ms	3.24ms	13.26ms
N. Virginia	63.48ms	11.98ms	4.87ms

Table 5.1: RTT latencies across different datacenters in ms.

5.7 Evaluation

In this section, we discuss QuORAM's experimental evaluations and contrast its performance with multiple baselines. Of particular interest is a baseline that resembles Obladi [47]'s approach to fault-tolerance. As noted earlier, to date Obladi is the only other ORAM-based system that tolerates trusted proxy failures. Obladi achieves this by relying on the fault tolerance guarantees of cloud databases; Obladi pushes the necessary state of the proxy periodically to the external fault tolerant database and recovers the proxy's state from the database if and when the proxy fails. While Obladi provides many additional guarantees, such as oblivious ACID transactional guarantees, we focus on its design choice for fault tolerance.

While replication forms the core of fault tolerance, the two systems choose contrasting designs to replicate data: Obladi relies on the external cloud database to manage the replicas and QuORAM manages the replicas itself. To precisely measure how the choice of replication affects performance we build a baseline consisting of a single TaORAM proxy (since TaoStore is the basis of QuORAM's ORAM scheme) that relies on a fault-tolerant open source database, CockroachDB [198], to replicate data. The goal of this baseline is to contrast the performance when an ORAM datastore (such as Obladi) relies on a replicated database for fault tolerance vs. using QuORAM.

5.7.1 Experimental Setup

We evaluated QuORAM and its baselines on AWS using r5.xlarge instances with 32GB of memory, Intel Xeon Platinum 8000 CPU with 4 cores @ 3.1GHz, and a gp2 SSD. Storage servers for QuORAM and its baselines persist the data on disk. We run our experiments on three different datacenters N. California, Ohio, and N. Virginia and Table 5.1 records the round-trip-time (RTT) latencies across and within the three datacenters. All the experiments place an ORAM unit (server & proxy) and a client process in each datacenter. Each client process creates 100 concurrent threads to achieve concurrency. We believe this reflects a setup for real-world applications where geo-distributed clients access data replicated across different datacenters. Note that we chose a replication factor of 3 as it is typically the default replication factor used in current state-of-the-art databases [53, 54].

Baselines:

Along with the CockroachDB-backed baseline, we evaluate QuORAM with 2 other baselines as well. Note that all baselines and QuORAM receive requests from geo-distributed clients. The 3 baselines are:

1. Insecure Replication Baseline: To measure the cost of providing obliviousness guarantees, we compare QuORAM with an insecure replication baseline system that deploys QuORAM's replication protocol (§5.6.1) for fault-tolerance. More precisely, a client queries from a majority quorum; for read operations it picks the value corresponding to the highest tag and for write operations it increments the highest tag and updates the value; it propagates the (potentially updated) tag and value to the same quorum it read from. In this baseline, the clients interact directly with the data store replicas, eliminating the need for proxies, and clients do not encrypt their data or perform any ORAM related operations.

2. Secure No Replication (TaoStore): To measure the costs and benefits of fault-tolerance, we use as a baseline the original non-replicated TaoStore [184] design consisting of a trusted proxy and an external server both located in N. California. We choose TaoStore as the non-replicated baseline over other concurrent ORAM schemes such as ConcurORAM [36] or Oblivistore [193] because QuORAM 's ORAM logic closely relates to TaoStore's and hence, TaoStore forms a better baseline for evaluating the costs-benefits of replication, without accounting for performance differences due to ORAM scheme disparities.

3. CockroachDB Baseline: This baseline uses TaoStore for obliviousness guarantees and CockroachDB [198] for fault-tolerance (via replication managed by CockroachDB). We use a single trusted proxy (analogous to Obladi's single-proxy design) placed in N. California and a three-node CockroachDB cluster with replicas distributed across N. California, Ohio, and N. Virginia data centers, similar to QuORAM's setup.

5.7.2 Implementation details

We implemented QuORAM as well as the three baselines by modifying an open-source Java implementation of TaoStore, which forms the base ORAM scheme of QuORAM. The implementation consists of ∼9,400 lines of Java code. To evaluate the systems, we use YCSB-like [44] benchmarking.

The storage server stores 1 GB of data with a block size of 4096 bytes and a bucket size of 4 blocks (i.e., each node in the tree stored at the external server consists of 4 blocks). To simulate an increasing load on the system, multiple client threads request logical read/write operations. By default, the experiments use 300 concurrent and geo-distributed clients accessing data at once (unless noted otherwise in an experiment). Each client chooses a type of operation at random, sends the request, waits for the response,

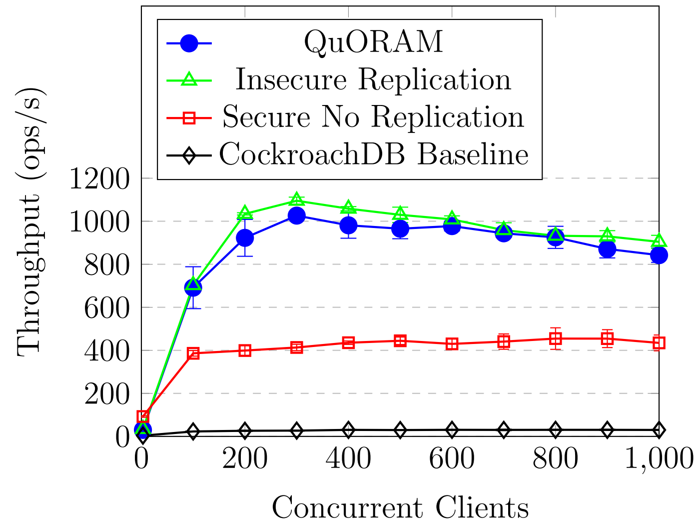

Figure 5.7: QuORAM's throughput is comparable with the Insecure replication baseline, 1.4x higher than the No Replication baseline, and 33.2x higher than using CockroachDB for fault-tolerance.

and then repeats the process. Each run of the experiment lasts three minutes, with all clients ending at precisely the same time. For each operation, the block to be read or written is chosen randomly among all the blocks using a Zipfian distribution with an exponent of 0.9 (unless stated otherwise in an experiment), and the operation type is picked uniformly at random between read and write. In all the experiments, each data point represents an average of 3 runs and also marks the confidence interval. For system configurations, we use a default value $k = 40$ and the daemon process accesses blocks every 100ms where blocks are selected in a pseudorandom order.

5.7.3 Throughput and Latency

In the first set of experiments, we compare the throughput and latency of QuORAM with the three baselines. Figures 5.7 and 5.8 respectively show throughput and latency observed while increasing the number of concurrent clients.

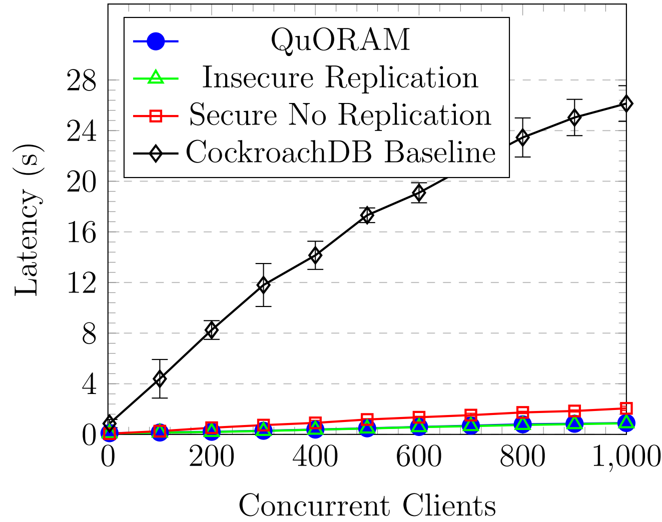

Figure 5.8: QuORAM's latency is comparable with the Insecure replication baseline, whereas the No replication baseline and CockroachDB suffer from a bottle-necked single proxy.

	Query phase	Propagate phase
QuORAM	12ms	0.55ms
Insecure Replication	0.05ms	0.03ms

Table 5.2: Processing time spent in the query and propagate phases by replicas in QuORAM vs. the Insecure replication baseline.

i. QuORAM vs. Insecure Replication Baseline

We first compare QuORAM with an insecure baseline that replicates data using QuORAM's replication protocol (§5.6.1). As seen in Figures 5.7 and 5.8, QuORAM's throughput and latency values are comparable with that of the insecure baseline in spite of QuORAM providing privacy and obliviousness guarantees. To better understand the minor performance differences between QuORAM and the insecure baseline, we measured the average processing times spent by a replica in both the query and propagate phases of the two protocols. Table 5.2 records the processing time breakdown. As noted in the table, QuORAM's query phase requires the most time because a proxy communicates with its server to fetch a path. This includes 3-6ms intra-datacenter communication la-

tency (Table 5.1). The proxy also decrypts the read path, merges it with the Subtree, and flushes the path, all of which incur processing latency. Meanwhile, the propagate phase merely updates a block in the Subtree. Although as noted in Table 5.2, the processing time for both phases of the insecure baseline require extremely low latency compared to QuORAM, the communication cost (Table 5.1) overwhelms the processing time of either protocols, causing both protocols to be latency bound. Due to this reason, both QuORAM and the insecure baseline have comparable performances. This experiment indicates that in geo-replicated datastores, the overhead of encrypting and hiding access patterns of data is negligible compared to communicating with geo-distributed replicas.

ii. QuORAM vs. Secure No Replication Baseline

When comparing QuORAM's performance with a non-fault tolerant baseline (TaoStore as-is), we see the most counter-intuitive result in this work. Because replication involves additional communication with replicas and maintaining additional data structures (e.g., incompleteCacheMap), one can expect a replicated solution to perform worse than its non-replicated counterpart. The reason why QuORAM outperforms a non-replicated TaoStore datastore is because TaoStore consists of a single proxy, located in N. California, which receives increasingly higher number of concurrent client requests, whereas the client load is balanced across the three proxies in QuORAM. More importantly, since the experiment consists of geo-distributed clients and the proxy resides in just one location, the clients farther from the proxy face large access latencies, reducing the overall performance. Due to both load balancing and geo-replication, QuORAM's peak throughput is **1.4x** higher than the non-replicated baseline.

iii. QuORAM vs. CockroachDB Baseline

Finally comparing QuORAM with a replicated ORAM scheme that relies on a fault-tolerant database, CockroachDB, both in-terms of throughput and latency, QuORAM clearly outperforms CockroachDB. The two main reasons causing CockroachDB to per-

form poorly are: (i) This baseline also consists of a single proxy that uses the read/write interface of CockroachDB to read and write the data on the external database. This single proxy, located in N. California, suffers from the same bottleneck issues as the non-replicated baseline. To mitigate the single proxy bottleneck, deploying multiple proxies – where a client communicates with any one proxy to access data – is a non-trivial task. This is because each access updates only one proxy's position map, stash, and subtree data structures, and the other proxies now have inconsistent data or position maps. Such solutions can neither guarantee linearizability nor obliviousness; (ii) The second reason causing CockroachDB to perform poorly is its choice of replication design: CockroachDB has a single leader for a given data item and this leader sequentially replicates data across replicas. Because of this single leader approach, since every read or write operation accesses the root node of the ORAM storage tree, *all* client operations are executed sequentially. QuORAM, on the other hand, employs a decentralized replication protocol, mitigating the single leader bottleneck. Because of the above two bottlenecks, CockroachDB performs worse with increasingly concurrent client requests.

5.7.4 Varying write-back threshold k

This set of experiments measures the throughput and latency of client accesses while varying the write-back threshold k, as seen in Figure 5.9. The parameter k resembles a batching threshold: the higher the value of k, the higher the number of paths written back together and vice versa. Although proxies in QuORAM process and maintain larger number of paths locally with higher k values, it also results in fewer write-backs. Moreover, because a background thread executes write-backs, k values do not have a significant impact on throughput (with a range of 980-1030 ops/sec) or latency (about 290 ms), as can be seen in Figure 5.9. This indicates the performance of the system is

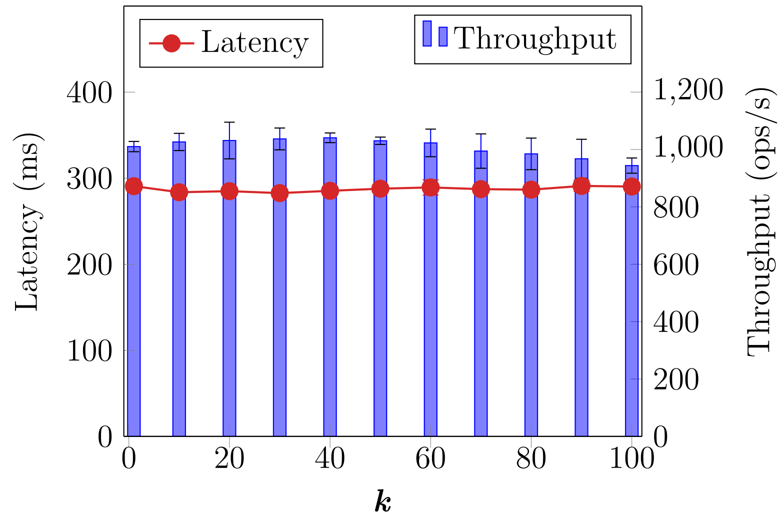

Figure 5.9: Varying the write-back frequency parameter k has no significant effect on throughput or latency of QuORAM.

independent of the frequency of write-backs.

5.7.5 Varying contention

This experiment measures QuORAM's performance – throughput and latency – while varying the contention levels in client generated workloads and the results are shown in Figure 5.10. Low contention, achieved by setting Zipfian exponent close to 0, implies clients select blocks uniformly at random from a pool of 262,140 blocks (the size of our dataset). High contention, achieved by setting Zipfian exponent to 0.9, indicates clients pick a small percent of the blocks (e.g., 10%) with a high probability. Typically, in non-oblivious datastores, contention in client workloads directly impacts the performance with higher contention causing low performance and vice versa. But the performance of an oblivious datastore, such as QuORAM, must remain independent of the contention in client workloads; otherwise an adversary can infer contention in client workloads just by observing requests served per second. As Figure 5.10 clearly indicates,

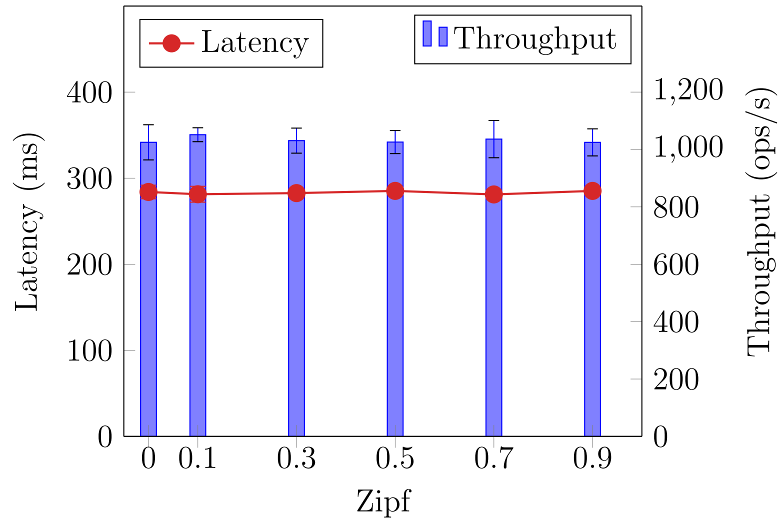

Figure 5.10: Varying Zipfian exponent to produce low to high contention workloads has no significant effect on throughput or latency of QuORAM.

QuORAM's throughput and latency values remain mostly constant with increasing contention (increasing Zipfian exponent) in client workloads. This experiment highlights the effectiveness of QuORAM in remaining impervious to contention in client workloads.

5.7.6 Stash and excessBlocks size analysis

In the next set of experiments, we measure the average sizes of Stash and excessBlocks data structures over a 10-second window, calculated for the duration of 6 minutes, as shown in Figures 5.11 and 5.12 respectively. Both figures depict the size of the respective data structures for two different Zipfian distributions in client workloads: Zipfian exponent close to 0 (≈ 0.00001) indicates low contention (i.e., most requests access unique blocks) and Zipfian exponent of 0.9 implies high contention (i.e., most requests access a small subset of blocks). Moreover, this experiment executes with the write-back threshold k set to 1. The reason we choose to analyze the sizes of Stash and excessBlocks with varying contention and with $k = 1$ is because of the memory issue discussed in §5.6.2.

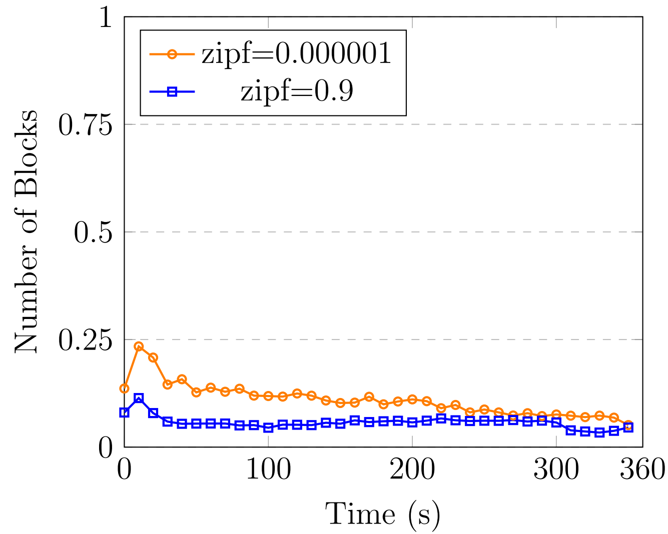

Figure 5.11: The number of blocks in Stash remains low. The Stash's 10-second moving average size is under 1 block (implies the Stash has at least one block in the last 10 seconds).

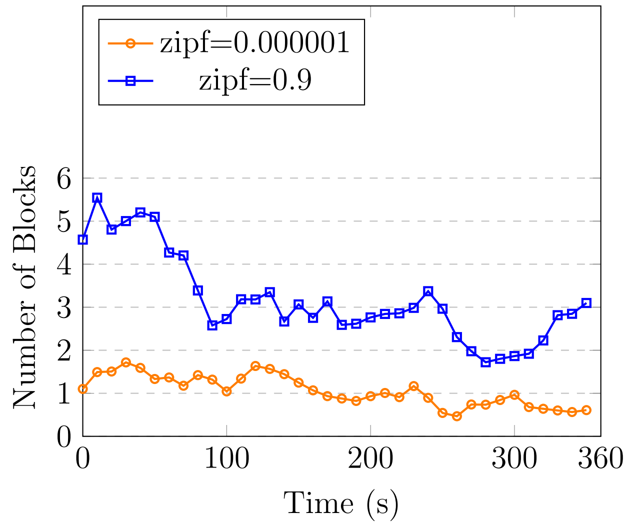

Figure 5.12: The number of blocks in excessBlocks remains low. The excessBlocks's 10-second moving average size at peak 6 blocks (= $0.33 \cdot logN$ where N=262140 blocks and $logN$ ≈18).

Recall that the memory issue is caused when say two logical operations access the same block and the second operation triggered a fake read. If the second operation's `o_write` arrives *after* the proxy initiates a write-back, the proxy cannot delete the block after receiving a write acknowledgement from the server (as TaoStore would have). To ensure the size of Subtree, which impacts the size of Stash, remains low, we move blocks that cause the memory issue into excessBlocks. Because excessBlocks's size can vary based on contention as well as when the write-back occurs frequently, we measure its sizes across two extreme contention values and the worst case write-back threshold. First, analyzing the Stash size, Figure 5.11 highlights that the size of the stash remains less than 1 over a 10-second window, matching QuORAM's theoretical Stash size guarantees of $logN$. Second, analyzing the size of excessBlocks, Figure 5.12 indicates that even though excessBlocks's size in larger for high contention, for both high and low contention workloads, it's size remains low (at worse $(0.33 \cdot logN)$ with N=262140 and $logN = 18$). We note that choosing various strategies of how the daemon process in a proxy accesses blocks – sequential, pseudorandom, or blocks from excessBlocks – has no significance on the size of excessBlocks. This experiment clearly highlights that both Stash and excessBlocks remain small for all types of contention in workloads.

5.7.7 Crash Experiment

The final experiment measures QuORAM's performance when one (N. California) of the three ORAM units crashes when 300 clients execute operations and the crashed unit remains unavailable for the remainder of the experiment. The throughput and latency over time is depicted in Figures 5.13 and 5.14 respectively. As the figures indicate, the throughput drops and the latency increases steeply as soon as the crash occurs; both values stabilize afterwards. In both figures, QuORAM's throughput stabilizes at $\sim$800

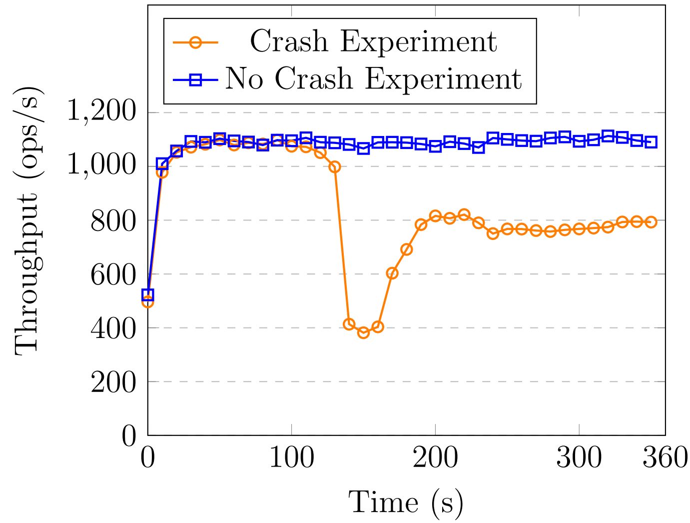

Figure 5.13: When an ORAM unit crashes, after a short adjustment period, throughput value stabilizes and the stabilized value is higher than the non-replicated baseline.

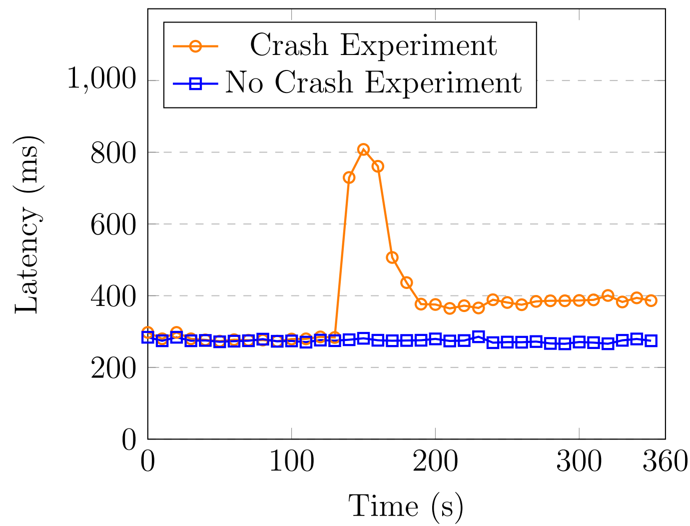

Figure 5.14: When an ORAM unit crashes, after a short adjustment period, latency value stabilizes and the stabilized value is lower than the non-replicated baseline.

ops/s and latency stabilizes at ∼400ms. Even when failures occur, QuORAM performs better that the non-replicated baseline. The drop in QuORAM's throughput, which is ∼300 ops/s, is roughly one-third of the overall throughput ∼1080 ops/s. In fact, the reason the drop in throughput is less than one-third of the total throughput (∼300 instead of ∼360) is because this experiment crashes the proxy in N. California, which adversely affects only one set of clients. Whereas the clients in Ohio and N. Virginia continue to benefit from forming a quorum of two nearby proxies (Table 5.1). This experiment shows that QuORAM performs better than the non-replicated baseline even while tolerating f ORAM unit failures.

5.8 Security of replicated ORAM datastores

This section discusses obliviousness of QuORAM. Recall the ORAM scheme and the security definitions defined in Section 5.5.2. While the underlying ORAM scheme TaORAM [184] is proved to be *aaob-secure* (adaptive asynchronous obliviousness), QuORAM extends *aaob-secure* definition to include *logical operations* and defines *l-aaob-security* in Section 5.5.2. Logical operations are client requested read and write operations, which may internally consist of ORAM read and write operations. *l-aaob-secure* is an indistinguishability based security definition defined using a security game $\mathcal{G}$ in Section 5.5.2.

Theorem 1: Assuming individual ORAM units are *aaob-secure*, QuORAM is *l-aaob-secure.*

Proof (Sketch): Sahin et al. proved the obliviousness of TaORAM in [184]. The most important property of TaORAM (and tree-based ORAMs in general) is that every logical access translates into fetching a random path from the server to the TaORAM Processor, right after the Processor receives the logical access request. TaORAM achieves this by initially randomly shuffling the dataset before uploading to the storage server, and

assigning a new random position to a block after each access. The position map in TaORAM's Processor keeps track of the random positions of all blocks.

We here focus only on the obliviousness of QuORAM, showing that it is *l-aaob-secure*. The security game $\mathcal{G}$ is defined in Section 5.5.2. Because the actual proof involves similar steps as TaORAM's, we omit the full proof due to lack of space but we outline the main steps necessary for the formal argument. The following are the key properties of QuORAM in arguing for its *l-aaob-security*:

1) During initialization, the game shuffles the data set D_b (after encryption) chosen by the adversary as done in TaORAM. Note that a consequence of this is that no two external servers store D_b in the same order.

2) QuORAM's replication protocol always accesses a quorum (majority) of ORAM units for the query phase and the same quorum for the propagate phase. An adversary $\mathcal{A}$ observing the communication between a client and the ORAM units sees 2 rounds of communication between the client and a quorum, for either type of logical operations, irrespective of the address or content of the block accessed.

3) In executing a logical operation, a proxy, p, is either part of the quorum or not. If p is part of the quorum, it always receives `o_read` before `o_write` (if `o_read` was dropped, the proxy sends negative acknowledgement for the `o_write`).

4) Given the fixed order of ORAM read and write requests for each logical request, in response to `o_read`, a proxy always fetches exactly one random path, either real or fake, from the server. There are three ways in which a path may become ready to be written back to the server. 1) The client sends an `o_write`, and then the path fetched for the corresponding `o_read` becomes ready to be written back. 2) The incompleteCacheMap becomes full and it chooses an entry to evict according to the eviction policy; the path associated with that entry becomes ready to be written back. 3) A path fetched by the daemon process is ready to be written back. When the number of paths to be written

back accumulates to k, the proxy writes them back in a batch. Importantly, the adversary can predict the trigger for each of the case above, since 1) it observes every `o_read` and `o_write` requests from the client and knows the random path fetched for each `o_read`, 2) it can deduce the entries that reside in incompleteCacheMap and when it becomes full and which entry should be evicted, and 3) the adversary predicts the access from the daemon process (based on the preset interval). Therefore, observing the write-backs to the server reveals no non-trivial information.

5) The incompleteCacheMap in QuORAM identifies blocks that are read but not yet written. Maintaining this information crucially avoids re-fetching a path from the server for a given logical request. Further, even if the incompleteCacheMap evicts an in-progress block, the proxy still retains the block locally until it is written back to the server.

6) If an adversary $\mathcal{A}$ crashes either a server or a proxy, especially in the middle of a query or a propagate phase, $\mathcal{A}$ observes the client, executing the protocol, randomly access another ORAM unit and send two sequential requests (query followed by propagate) to this additional unit.

7) The game notifies the completion of a logical operation to the adversary only *after* a quorum of ORAM units complete executing both the query and propagate phase. If the adversary delays scheduling one or more messages in either of the phases, it receives delayed notification from the game.

In the security game (defined in game $\mathcal{G}$ in §5.5.2), an adversary generates two data sets of the same size D_0 and D_1 and schedules multiple but finite pairs of logical requests $(lop_{0,m}, lop_{1,m})$, where m identifies each request pair generated by the adversary. The game randomly picks the challenge bit $b \in \{0,1\}$ and stores only D_b in QuORAM and executes only $lop_{b,m}$ from each request pair. To store D_b in QuORAM, the game calls Rep-ORAM on D_b by invoking $D^b_{encK_i}, K_i \leftarrow \mathsf{Encode}_i(D_b)$ for each ORAM unit i. The external server and the proxy of an ORAM unit i store the encrypted data D_{encK_i} and

the secret key K_i, respectively. The game executes QuORAM's replication protocol as defined in §5.6.1 for each logical request $lop_{b,m}$. The adversary does not see the output value of any operation it schedules (if it did, it would be trivial to guess the challenge bit). To prove that QuORAM is *l-aaob-secure*, we need to argue that an adversary has negligible advantage over randomly guessing the value of challenge bit b.

To do this, we show that from the adversary's point of view, it cannot distinguish a real execution of the game with a simulated game that does not use D_b or $lop_{b,m}$ for either b. First, instead of storing D_b, the simulated game stores encryption of dummy blocks (e.g., zero-value) and replaces block values in each $lop_{b,m}$ logical request also with encryption of dummy blocks. Next, it simulates the view of the adversary as follows:

(i). For each `o_read` request, a quorum (majority) of ORAM unit proxies are accessed; (ii). After the first access, the proxies always fetch one random path from the server and upon receiving the server response, proxies send a (response) message back; (iii) For each `o_write` request, the same quorum of ORAM unit proxies are accessed the second time, and they return to the client a small (acknowledgement) message; (iv) The simulator keeps track of the paths that are ready to be written-back triggered by `o_write`, as well as entries evicted from the incompleteCacheMap and accesses by the daemon process, and batch-write k paths back to the server, whenever k paths become ready.

Based on the above discussed properties of QuORAM, we assert that the adversary cannot distinguish the access behavior in the real and simulated cases, even in the presence of crash failures. This implies the *l-aaob-secure* of QuORAM.

5.9 Linearizability

As noted in TaoStore [184], the correctness of a read or write operation differs from the obliviousness of the operation. Similar to TaORAM [184], QuORAM defines correct-

ness using linearizability or *atomic semantics*: to an external observer, a client operation appears to take effect at a specific instance between the operation's invocation and its response indicating the operation's success. This section proves the correctness of QuORAM.

To argue for the correctness of QuORAM, we use the game $\mathcal{G}$ defined in Section 5.5.2 where the adversary schedules logical read/write operations but with a slight modification where the adversary now receives the response values and hence the challenge bit is non-existent. We call the modified game $\mathcal{G}_{corr}$ and use it in arguing correctness.

Definitions: A history Hist represents a sequence of logical read/write operations, viewed as the transcript after executing game $\mathcal{G}_{corr}$. Each operation op_i in Hist consists of an invocation event inv_i and a response event resp_i (which occurs after a successful propagate phase in QuORAM). A history is said to be complete if for every invocation event inv_i in the history there exists a corresponding response event resp_i; and otherwise the history is said to be partial.

We represent each operation op_i as $(op_{id}, bId, tag_i, v_i, u_i)$ where op_{id} identifies a globally unique logical operation, bId identifies a data block, tag_i represents a non-decreasing tag associated with the block, v_i equals $\perp$ for read operations and otherwise block's value to be updated with, and u_i indicates the existing value of the block prior to executing op_i, derived by a client after the query phase of op_i.

Similar to [184], $\leq_{lin}$ defines a linearizable relation between any two operations op_i and op_j: $op_i \leq_{lin} op_j$ implies resp_i precedes inv_j in a given history. We note that linearizability is defined for a single data block, i.e., both op_i and op_j operate on the same block bId. Given a complete and finite history of operations executed by QuORAM, this section proves QuORAM is linearizable, provided any adversary $\mathcal{A}$ eventually delivers all messages (after delaying and/or reordering).

Lemma 1: A block bId's response value u_i, derived by a client after a successful query

phase of an operation op_i, corresponds to bId's highest tagged value.

Proof: Since each logical request in QuORAM reads from and writes to a (majority) quorum, there exists at least one over-lapping ORAM unit between any two logical requests. For each ORAM unit, TaORAM [184] guarantees that the unit maintains fresh-subtree invariant: "The contents on the paths in the local subtree and stash are always up-to-date, while the server contains the most up-to-date content for the remaining blocks". Thus, when a client executes the query phase of a logical operation op_i, at least one ORAM unit answers with block bId's value u_i corresponding to the highest tag (either from the ORAM unit's proxy or the server), proving Lemma 1 holds. □

Lemma 2: Tags of a block *bId* maintained by an ORAM unit (either at the proxy or at the server) are monotonically non-decreasing.

Proof: As described in Algorithm 7, clients in QuORAM either retains tag values (for reads) or increments them (for writes) but never decrements tag values. Lemma 1 shows that a client always receives the highest tag for a block while executing the query phase, which it may retain or increment based on the type of the operation. Further, as discussed in §5.6.1, an ORAM unit's proxy updates a block's tag after receiving an `o_write` request *if and only if the new tag is greater than the block's current tag.* Based on the above arguments, it is shown that Lemma 2 holds. □

In our proposed system, linearizability captures two main relations between any two operations in a history: (i) the tag values of any two completed logical operations have a strict $<$ or $\leq$ relation; and (ii) a given logical operation – read or write – is atomic. The former point captures the relative ordering of logical operations. The latter point implies that if an operation op_i wrote a block, then an operation op_j immediately succeeding op_i must read the block written by op_i; and if operation op_i merely read a block without writing it, then operation op_j immediately succeeding op_i must also read the same value

as op_i. We formally define the two relations captured by linearizability as follows.

Definition 1: A complete and finite history Hist is linearizable if for any two logical operations $op_i = (bId, tag_i, v_i, u_i)$ and $op_j = (bId, tag_j, v_j, u_j)$, and op_i, $op_j \in$ Hist, the following conditions hold:

❶ if op_i precedes op_j, then (i) $tag_i < tag_j$ if op_j is a write operation, or (ii) $tag_i \leq tag_j$ if op_j is a read operation.

❷ if op_i precedes op_j such that tag_i is the highest tag less than or equal to tag_j, then (i) $u_j = v_i$ if $v_i \neq \perp$ (op_i is a write), or (ii) $u_j = u_i$ if $v_i = \perp$ (op_i is a read).

Theorem 2: QuORAM provides linearizability.

Proof: ❶ To prove the first condition, we consider the two possible types of operations op_j can be:

(i) *If op_j is a write*: From Lemma 1 and 2, a logical write always increments the highest tag of a block. Since op_j is a write, and op_i may or may not be, due to the quorum intersection, op_j receives the highest tag in its query phase and increments it. Hence, the tag of op_j is strictly greater than that of op_i.

(ii) *If op_j is a read*: From Lemma 1 and 2, given the tag of a block is monotonically non-decreasing, we know that $tag_j \not< tag_i$, as op_i precedes op_j. Since tags are incremented only on writes, if no write took place between op_i and op_j, then $tag_i = tag_j$; whereas if a write operation op_k occurred after op_i and before op_j, then $tag_i < tag_k$ (from step (i)), and by transitivity, $tag_i < tag_j$. This is true for any number of write operations between op_i and op_j. Hence, $tag_i \leq tag_j$.

❷ Given that tag_i is the highest tag less than or equal to tag_j, irrespective of the type of operation of op_j, due to Lemma 1, when op_j executes the query phase, it receives the current highest tag of the block, i.e., tag_i and its associated value. (i) Now, if op_i wrote the block, then the block's value is v_i and hence when op_j queries the block, it

receives v_i. Thus $u_j = v_i$. This shows that writes are atomic as any operation executing after a write reads the updated value.

(ii) If op_i merely read the value, which was equal to u_i, then since op_j immediately succeeds op_i for block *bId*, op_j's read value also equals u_i as no other operation updated the block. Thus $u_i = u_j$. This shows that reads are atomic. □

5.10 Space analysis

This section analyzes the stash size of QuORAM and the space utilized at the proxy.

5.10.1 Stash size analysis

Lemma 3: Similar to TaORAM, QuORAM's stash size is bounded by any function *F(N)* $= \omega \cdot logN$, except with negligible probability in N.

Proof: The core idea of this proof lies in mapping the execution of QuORAM to that of TaORAM in a straight-forward way. TaORAM's stash size is proved to be bounded by a function $F(N) = \omega \cdot logN$ (e.g., $F(N) = (logloglogN) \cdot logN$) and by mapping QuORAM's execution to that of TaORAM we prove that QuORAM has the same stash size guarantees as TaORAM.

To analyze QuORAM's stash size, recall the details of the unbounded space issue and its solution discussed in §5.6.2. The memory issue is caused due to the asynchrony in receiving `o_read` and `o_write` requests for a logical request; if a proxy initiates a write-back in between receiving the two requests, and if the `o_read` had triggered a fake read, the proxy cannot delete the block after receiving a write acknowledgement from the server. This is because the block's latest `o_write` arrived *after* the proxy initiated the write-back. In the unlikely case that this block or any block in its path is never accessed again, this block will always reside in the Subtree. This may in-turn affect the

size of the Stash. QuORAM mitigates this issue by moving such blocks to excessBlocks datastructure and the daemon process in each proxy accesses (i.e., mimics `o_read`s and `o_write`s) blocks in the excessBlocks at pre-set intervals of time. This can be viewed as, from TaORAM's perspective, all blocks that can be deleted after receiving a write-back acknowledgement from the server will be deleted from the Subtree (and some may move to excessBlocks). As seen with this abstraction, QuORAM relies on TaORAM's logic of freeing the Subtree, without any changes, and hence QuORAM's stash size analysis follows that of TaORAM and the size is bounded by any function $F(N) = \omega \cdot logN$, except with negligible probability in N. □

5.10.2 Proxy space analysis

A proxy in QuORAM maintains two types of information: temporary data pertaining to the on-going requests and permanent data related to the ORAM scheme. As discussed in TaoStore [184] and as proved in the Lemma 3, with regard to permanent data for the ORAM scheme, the proxy maintains a stash (of size $\omega \cdot logN$), position map (of size $N \cdot logN$), and secret key (of size λ), and so the order of storage size of this is:

$$ORAM\ related = O(N \cdot logN + \lambda)$$

The size of the temporary data is directly proportional to the number of concurrent logical requests, I. The size of I depends on many dynamically changing parameters such as the number of concurrent clients and their request sending rate, the processing powers of the proxy, the server and the clients, the bandwidth and asynchronous nature of the network, the geo-graphical distance between the client and the ORAM units, etc. The temporary storage, analyzed w.r.t I, stems mainly the incompleteCacheMap and excessBlocks. The incompleteCacheMap has bounded size fixed by a configurable

parameter, R, and the excessBlocks datastructure's size depends on the access pattern as well as the daemon process's data access interval. Since we have already shown through experiments that excessBlocks' size remains a constant, E, in most practical workloads, the temporary storage can be computed as:

$$Temporary\ space = \underbrace{O(I \cdot logN)}_{Subtree} + \underbrace{O(R)}_{incompleteCacheMap} + \underbrace{O(E)}_{excessBlocks}$$

We consider steady execution state to be the one where the server responds to the path fetching requests and the clients send `o_write` requests, both within a reasonable time bounds. In steady state, after receiving `o_write` request for a given logical request, the proxy removes the incompleteCacheMap entry and writes back to the server after k logical requests, freeing Subtree. Hence the steady state memory consumption of QuORAM is the same as TaORAM:

$$Steady\ memory\ use = O(k \cdot logN + N \cdot logN + \lambda)$$

A malicious adversary can hamper the steady state in two ways: (i) not send any responses to the path fetch requests, or (ii) discard all `o_write` requests from clients. The proxy mitigates the latter case (as discussed in §5.6.2) by evicting entries corresponding to previously received `o_reads` from incompleteCacheMap; evictions increment the *paths* counter and in the discussed adversarial case, a proxy writes k paths back after k evictions (if a proxy receives no `o_write`). For the former case, if a proxy receives no path fetch response for a set threshold of time, the proxy stops accepting any requests from a client.

5.11 Related Work

While the literature on ORAM schemes consists of many works [78, 194, 192, 193, 23, 184, 136, 47], to date, Obladi [47] by Crooks et al. is the only system to consider the fault-tolerance aspect of an ORAM system. While Obladi provides transactional (ACID) guarantees in an ORAM setting, it compares to QuORAM in its *durability* or fault-tolerance aspect. Obladi assumes the external and untrusted cloud storage server to be inherently fault-tolerant – a property guaranteed by most cloud providers – and relies on this guarantee to make the ORAM proxy fault-tolerant as well. Obladi pushes the state of the stateful proxy to the external server at periodic intervals; if the proxy crashes, it is restored to the last state pushed to the server. QuORAM has two main advantages over Obladi's design choice of fault-tolerance: i) in spite of backing up the proxy's state at set intervals, Obladi becomes unavailable *during* proxy failures and recovery, and ii) as shown in the experiments, relying on cloud providers for fault-tolerance incurs performance penalties compared to QuORAM's choice of fault-tolerance. Another work EHAP-ORAM [135] relies on Non-volatile Memory (NVM) based hardware to persist data to recover from crashes. But the proposed solution cannot be generalized for non-NVM based ORAM datastores.

In Pharos [223], Zakhary et al. are one of the first to demonstrate the challenges of extending ORAM schemes to include replication. The authors show that naively replicating an ORAM system leaks non-trivial sensitive information. However, no correct ORAM fault-tolerant solution is proposed.

In a separate line of work, many works [36, 194, 192, 138, 136, 193, 228] have looked at extending a single ORAM server model to multi-server, multi-cloud settings. In SSS-ORAM[194] Stefanov et al. propose partitioned ORAM: an ORAM of N items split into $\sqrt{N}$ ORAMs, each of $\sqrt{N}$ size, albeit with a single cloud assumption. In [138], Lu et

al. propose a distributed two-server ORAM from a theoretical perspective. They show that with two non-colluding servers, client bandwidth can be reduced to O(logN). In [192] Stefanov et al. extend [194] to propose a multi-cloud oblivious storage solution to reduce client-cloud bandwidth cost. The paper discusses a 2-cloud solution: an ORAM of N items is split across two non-colluding servers where after each data block's access, the two servers perform *two-cloud shuffling* to randomly shuffle the accessed block before its next access. In [136] Liu et al. build on [192] to optimize not only client storage and server bandwidth, but also on the cloud-cloud bandwidth, leading to reduced overall response time. Oblivistore[193] by Stefanov et al. also extends SSS-ORAM [194] to not only incorporate asynchronous concurrency but also to distribute an N item ORAM into multiple servers. The work also proposes ways to dynamically add ORAM nodes and external storage servers. CURIOUS [23] proposes a simpler solution to distribute data across multiple storage servers and serves concurrent client requests. ConcurORAM [36] allows a constant c number of concurrent clients to query at a time and require APIs for fine-grained locking and additional datastructures from the server.

While the above works extend a partition-based ORAM scheme ([194]) to multi-server or multi-cloud schemes, in [228] Zhang et al. extend the tree-based ORAM ([195]) into a two-server setting by splitting the storage tree across two non-colluding servers to enhance performance. While the above proposals distribute data across storage servers, their deployment uses a single proxy. Recently Snoopy [50] partitions the data *and* the proxies where for scalability, proxies executing on trusted hardware serve different sets of client requests.

The main differences between prior proposals [194, 192, 138, 136, 193, 228, 50] and QuORAM are: i). the former proposals are non-replicated, i.e. each server stores a disjoint set of data items, whereas in QuORAM all servers store the same set of data items; ii) the former proposals are not fault-tolerant and can lose the data if a server

or an ORAM client fails, unlike in QuORAM that tolerates server and ORAM client failures.

5.12 Conclusion

This work proposed QuORAM a quorum-replicated ORAM datastore that provides fault-tolerance and linearizable semantics. To date, QuORAM is the first system to replicate data while preserving obliviousness by hiding access patterns. QuORAM's novel replication protocol avoids locking – a standard technique to guarantee linearizability in distributed data systems – as employing locking can leak non-trivial information. Because QuORAM's replication protocol chooses a decentralized design, QuORAM performs **33.2x** better in throughput compared to relying on CockroachDB for fault tolerance, which consists of a centralized replication protocol. QuORAM's evaluation with a non-replicated ORAM baseline establishes the performance benefits of replication: due to geo-replication, clients can access data from close-by replicas thus increasing QuORAM's peak throughput by **1.4x** compared to the non-replicated baseline. Finally, the experiments indicate that the overhead of achieving obliviousness using QuORAM is negligible compared to the cost of fault-tolerance due to communication among geo-distributed replicas.

Chapter 6

ORTOA: One Round Trip Oblivious Access

Cloud based storage-as-a-service is quickly gaining popularity due to its many advantages such as scalability and pay-as-you-use cost model. However, storing data using third-party services on third-party servers creates vulnerabilities, especially pertaining to data privacy. While data encryption is an obvious choice to achieve data privacy, attacks based on access patterns have shown that mere encryption is insufficient to fully hide the data from the storage vendor. Solutions such as Oblivious RAM (ORAM) and Private Information Retrieval (PIR) propose techniques to hide data access patterns. Hiding access patterns involves hiding both the specific data item accessed and the type of access – read or write – on the item. Most existing obliviousness solutions focus on hiding the accessed data item; whereas to hide the type of access, these techniques communicate twice with the remote storage, sequentially: once to read and once to write, even though one of the rounds is redundant with regard to a user's access request. To mitigate this redundancy, we propose ORTOA- a One Round Trip Oblivious Access protocol that reads or writes data stored on remote storage *in one round without revealing the type*

of access. To our knowledge, ORTOA is the first generalized protocol to obfuscate the type of access in a single round, reducing the communication overhead by half. ORTOA focuses on hiding the type of access and due to its generalized design, can be integrated with many existing obliviousness techniques that hide the specific data item accessed. Our experimental evaluations show that compared to ORTOA a baseline that requires two rounds to hide the type of access incurs **0.76x-1.61x** higher latency and **43%-61%** lower throughput than ORTOA.

6.1 Introduction

Data privacy is becoming one of the major challenges faced by the systems community today. Industries such as Facebook and Google are fined millions to billions [60, 84] of dollars for violating user data privacy, creating an urgent need to build systems that provide data privacy. While *data encryption* provides a preliminary data privacy solution, many works [166, 165, 105, 211] highlight that mere data encryption is insufficient to preserve data privacy. These works show that just by observing data access patterns, an adversary can infer non-trivial information about the data or the user. Such inference attacks are termed *access pattern attacks*.

Solutions such as Oblivious RAM (ORAM) [78, 195, 25, 184, 181, 215, 51] and Private Information Retrieval (PIR) [42, 218, 120] provide mechanisms to hide access patterns. Most of these works assume trusted clients who host their data on untrusted servers that are typically managed by third party cloud providers. ORAM solutions achieve access pattern obliviousness by shuffling the physical locations of the data stored on the untrusted server after every access. While some PIR schemes write the data [25, 134], most PIR schemes focus on retrieving or reading a data item without revealing the identity of the retrieved item to the external server. More recently, Pancake [90] proposed

frequency smoothing to obfuscate access patterns wherein the access frequencies to all entries in the database are smoothed so that an adversary cannot infer any non-trivial insights on the data. Albeit using a less stringent but realistic security model, Pancake highlights the highly pragmatic nature of frequency smoothing, with significantly better performance than ORAM based solutions.

In general, access pattern obliviousness in the above schemes consists of two aspects: (i) hiding the exact data item, or rather the exact physical location of the data item accessed by a client; (ii) hiding the *type* of access, i.e. a read vs. a write, requested by a client. To our knowledge, most existing solutions for access pattern obliviousness focus on proposing novel ways to solve aspect (i); whereas for aspect (ii), the most commonly adapted solution is to always perform a read followed by a write [195, 184, 90, 181], irrespective of the type of request. Always reading followed by writing to hide the type of access incurs *two sequential rounds* of accesses between the clients and the external server resulting in significant overhead; *eliminating this additional overhead is the focus of this chapter.*

The goal of this work is to provide a one-round solution to read or write data stored on an external server *without revealing the type of access*. This work does not focus on hiding the physical locations that clients access, which is orthogonal to hiding the type of access. The proposed solution can be integrated with solutions such as frequency smoothing or ORAM to hide the specific data item accessed in a single round.

We propose, ORTOA, a novel One Round Trip Oblivious Access protocol to access a data item stored on an external untrusted server without revealing the type of access, *in a single round*. This reduction in one round of communication plays a vital role in reducing end-to-end latency, especially in geo-distributed settings. For companies such as Amazon and Google, end-to-end latency directly impacts revenue. For example, Amazon

reported losing 1% revenue (worth $3.8 billion!) for every 100 ms lag in loading pages [6]; Google stated that its traffic drops by 20% if search results take an additional 500 ms to load [86].

Our motivation in proposing ORTOA is to bridge the gap between traditional vs. privacy-preserving datastores. Contrasting oblivious datastores with their trusted non-privacy-preserving counterparts, many real world databases such as MongoDB [159] and Redis [180] read and write (or get and put) data in a single round. The low performance of oblivious datastores forms a major barrier towards their adoption in industry. Hence, by proposing a technique that allows oblivious datastores to read/write data in a single round trip, we aim to bridge some of the gaps prevalent in commercializing oblivious datastores.

Furthermore, with increasing privacy laws such as GDPR [72] that prohibit data movement across continents, a solution such as ORTOA becomes even more relevant. This is because cross-continent communication suffers from expensive latency and avoiding sequential rounds of communication can be of great value for application developers. With restricted data movement, we believe that new protocols should trade-off sending larger amounts of data for reduced number of communication rounds.

Related work: To the best of our knowledge, ORTOA is the only solution that tackles the problem of hiding the type of operation in a generalized manner. The literature on ORAM constructions consists of a number of specialized solutions that achieve single round communication complexity [215, 139, 63, 51, 77, 26, 74, 70]. Despite achieving a one-round ORAM scheme, these solutions primarily differ from ORTOA in that they mainly focus on hiding the data access patterns, with mechanisms to hide the type of access that is tightly coupled with hiding access pattern. ORTOA on the other hand focuses on hiding the type of access in a more *generalized* way that can be adapted to construct obliviousness solutions such as ORAM or frequency smoothing [90]. Moreover,

all of the above single round ORAM schemes have a lower bound bandwidth cost of *log(N)*, where N is the number of data items [78, 130] or $\sqrt{N}$ lower bound when the data storage server performs no computations [33]. Most of this cost stems from hiding the data access patterns. Since ORTOA focuses only on obfuscating the type of access, it has a constant bandwidth cost independent of N, the number of data items (as will be shown in §6.4).

Chapter organization: The chapter is organized as follows: Section 6.2 presents the system and security model; Section 6.3 proposes a one-round oblivious access solution using an existing cryptographic primitive, fully homomorphic encryption, and discusses the impracticality of this approach. Section 6.4 then presents our novel protocol, ORTOA, to obliviously read or write in one round, followed by Section 6.5 discussing a space optimization and other optimizations of the protocol. Section 6.6 presents an experimental evaluation of ORTOA. Section 6.7 proves the security of ORTOA and finally Section 6.8 concludes the chapter.

6.2 System and Security Model

6.2.1 System Model

ORTOA is designed for key-value stores where a unique key identifies a given data item and the key-value store supports single key `GET` and `PUT` operations. The data is stored on an external server(s) managed by a third party, analogous to renting servers from third party cloud providers.

We assume the external server that stores the data to be untrusted. Furthermore, the system uses a proxy model commonly deployed in many privacy preserving data systems [179, 184, 90, 193]. The proxy is assumed to be trusted and the clients interact with the

external server by routing requests through the proxy. Alternately, the system can also be viewed as a single trusted client interacting with the externally stored data on behalf of users from within the trusted domain. The proxy is a stateful entity and remains highly available; ensuring high availability of the proxy is orthogonal to the protocol presented here.

All communication channels – clients to proxy, proxy to server – are asynchronous, unreliable, and insecure. The adversary can view (encrypted) messages, delay message deliveries, or reorder messages. All communication channels use encryption mechanisms such as transport layer security [205] to mitigate message tampering.

6.2.2 Data and Storage Model

Each data item consists of a unique key and a value, where all values are of equal length – an assumption necessary to avoid any leaks based on the length of the values (equal length can be achieved by padding). Neither an item's key nor its value is stored in the clear at the server. For a given key-value item $< k, v >$, the keys are encoded using pseudorandom functions (PRFs) [1]. A PRF's determinism permits a proxy to encode a given key multiple times while resulting in the same encoding; this encoding can then be used to access the value of a given key from the server. We use a procedure Enc to encode the values (this procedure differs from Section 6.3 to Section 6.4). For a key k and its corresponding value v, the server essentially stores $< PRF(k), Enc(v) >$.

6.2.3 Threat Model

As mentioned earlier, this work focuses on hiding only the type of access generated by clients and *not the actual physical locations accessed by client requests.* We assume an

[1] Alternate to PRFs, searchable encryption schemes can also be used. The main requirement is to have a deterministic encoding of plaintext keys.

honest-but-curious adversary that wants to learn the type of access performed by clients without deviating from executing the designated protocol correctly. The adversary can control the external server as well as all the communication channels – proxy to external server and clients to proxy. We further assume the adversary can access (encrypted) queries to and from a sender and can inject queries (say by compromising clients).

6.3 FHE based solution

This section presents a one round mechanism to hide the type of accesses using an existing cryptographic primitive, Fully Homomorphic Encryption (FHE) [73, 30, 61]. We first provide a high-level overview of FHE, and then present a one-round read-write solution that uses FHE, and finally discuss the impracticality of the solution.

6.3.1 Fully Homomorphic Encryption (FHE)

Homomorphic encryption is a form of encryption scheme that allows computing on encrypted data without having to decrypt the data, such that the result of the computation remains encrypted. Homomorphic encryption schemes add a small random term, called *noise*, during the encryption process to guarantee security. A homomorphic encryption function $\mathcal{HE}$ takes a secret-key sk, a message m, and a noise value n as input and produces the ciphertext, ct, as output as shown in Equation 6.1. The corresponding decryption function $\mathcal{HD}$ takes the secret-key and the ciphertext as input to produce message m:

$$ct = \mathcal{HE}(sk, m, n); \qquad m = \mathcal{HD}(sk, ct) \tag{6.1}$$

An important property of a homomorphic encryption scheme is that the noise must be small; in fact, the decryption function fails if the noise becomes greater than a threshold

value, a value that depends on a given FHE scheme.

Homomorphic encryption schemes allow computing over encrypted data. Some homomorphic encryption schemes support addition [173, 21] and some other schemes support multiplication [58]. A fully homomorphic encryption (FHE) scheme supports both addition and multiplication on encrypted data [73, 30, 61]. An FHE scheme, $\mathcal{FHE}$, applied on two messages $m1$ and $m2$ (and two noise values $n1$ and $n2$) can perform the following two operations:

$$\mathcal{FHE}(m1; n1) + \mathcal{FHE}(m2; n2) = \mathcal{FHE}(m1 + m2; n1 + n2) \tag{6.2}$$

$$\mathcal{FHE}(m1; n1) * \mathcal{FHE}(m2; n2) = \mathcal{FHE}(m1 * m2; n1 * n2) \tag{6.3}$$

For small noise values $n1$ and $n2$, decrypting $\mathcal{FHE}(m1 + m2; n1 + n2)$ results in the plaintext addition of $m1 + m2$, and similarly decrypting $\mathcal{FHE}(m1 * m2; n1 * n2)$ results in the plaintext multiplication of $m1 * m2$. As illustrated above, each homomorphic operation increases the amount of noise included in the encrypted value.

6.3.2 One-round oblivious read-write using FHE

To hide the type of client operation, i.e., read or write, from an adversary who might control the storage server, it is necessary for both read and write requests to be indistinguishable. Hence, both operations need to read *and* write a given physical location. More specifically, a read request should write back the same value it read, while a write request should write the new value, potentially distinct from the value it read. This is especially challenging to achieve in a single round *as the value to be read is stored only at the external server*; due to this challenge, existing solutions communicate with the server twice: first to read an item and then to write it.

We aim to use FHE to support executing read and write operations in a single round of communication to the external key-value store. Specifically, this section uses an FHE scheme as the encoding procedure *Enc* specified in Section 6.2.2 to encrypt the values of the key-value store. For a given key-value pair, the server stores $< PRF(k), \mathcal{FHE}(v) >$.

Let v_{old} be the current value of a given data item, which is stored only at the server (after encrypting $\mathcal{FHE}(v_{old})$), and let v_{new} be the updated value of the data item, for a write operation (and an 'empty' value for a read). The challenge is to develop an FHE procedure (or computation) PR with parameters $\mathcal{FHE}(v_{old})$ and $\mathcal{FHE}(v_{new})$ such that:

$$
\begin{aligned}
For\,reads: \textsc{Pr}(\mathcal{FHE}(v_{old})\ ,\ \mathcal{FHE}(v_{new})) &= \mathcal{FHE}(\boldsymbol{v_{old}}) \\
For\,writes: \textsc{Pr}(\mathcal{FHE}(v_{old})\ ,\ \mathcal{FHE}(v_{new})) &= \mathcal{FHE}(\boldsymbol{v_{new}}) \qquad (5)
\end{aligned}
$$

With such a procedure, the external server can execute the same procedure PR for both read and write requests but the result of PR would vary depending on the type of access. If we can design such a procedure, since the server already stores $\mathcal{FHE}(v_{old})$, the proxy only needs to send $\mathcal{FHE}(v_{new})$ in a single round and expect the correct result for either type of operations.

To develop such a procedure, the proxy creates a two-dimensional binary vector $\mathcal{C} = [c_r, c_w]$ where c_r is 1 for read operations (otherwise 0) and c_w is a 1 for write operations (otherwise 0). To see how the vector can be helpful, briefly disregard any data encryption and consider the data in the plain. We construct a procedure PR′:

Procedure $\textsc{Pr}'(v_{old}, v_{new}, [c_r, c_w])$:

RETURN $(v_{old} * c_r) + (v_{new} * c_w)$

For reads, when $c_r = 1$ and $c_w = 0$, the result of PR′ is v_{old}; otherwise, for writes when $c_r = 0$ and $c_w = 1$, the result of PR′ is v_{new}. The above procedure gives us the

desired functionality, albeit with no encryption. Given that FHE encrypted values can be added and multiplied, $\textsc{Pr}'$ can be refined to procedure $\textsc{Pr}$ to include FHE encrypted inputs:

Procedure

$\textsc{Pr}(\mathcal{FHE}(v_{old}), \mathcal{FHE}(v_{new}), [\mathcal{FHE}(c_r), \mathcal{FHE}(c_w)])$:

RETURN $\mathcal{FHE}(v_{old}) * \mathcal{FHE}(c_r) + \mathcal{FHE}(v_{new}) * \mathcal{FHE}(c_w)$

With Procedure $\textsc{Pr}$ that results in the desired outcomes as defined in Equation 5, the next steps elaborate on the specific operations of the proxy and the server:

(1) Upon receiving either a `Read(`k`)` or a `Write(`k, v_{new}`)` request from a client, the proxy creates vector $\mathcal{C}$ such that for reads, $\mathcal{C} = [1, 0]$ and for writes, $\mathcal{C} = [0, 1]$.

(2) Proxy then sends $\mathcal{FHE}(\mathcal{C})$, i.e. $[\mathcal{FHE}(c_r), \mathcal{FHE}(c_w)]$, along with $\mathcal{FHE}(v_{new})$, where v_{new} has a dummy value for reads. It also sends $PRF(k)$ so that the server can identify the location to access.

(3) The server, upon receiving the encoded key along with the 3 encrypted entities, reads the value currently stored at key $PRF(k)$. The server then executes Procedure $\textsc{Pr}$ by using the stored value $\mathcal{FHE}(v_{old})$ and the 3 entities sent by the proxy. The server then updates its stored value to the output of the computation and sends the output back to the proxy.

(4) Given that either c_r or c_w is 0, Procedure $\textsc{Pr}$'s output will either be $\mathcal{FHE}(v_{old})$ for reads or $\mathcal{FHE}(v_{new})$ for writes, as shown in Equation 7:

$$\begin{aligned} For\,reads: \mathcal{FHE}(v_{old}) * \mathcal{FHE}(1) \ + \ \mathcal{FHE}(v_{new}) * \mathcal{FHE}(0) = \mathcal{FHE}(v_{old}) \\ For\,writes: \mathcal{FHE}(v_{old}) * \mathcal{FHE}(0) \ + \ \mathcal{FHE}(v_{new}) * \mathcal{FHE}(1) = \mathcal{FHE}(v_{new}) \end{aligned} \tag{7}$$

For reads, the proxy decrypts $\mathcal{FHE}(v_{old})$ using FHE's secret-key to retrieve the data

item's value. For writes, the proxy ignores the returned value.

Thus, by leveraging the properties of FHEs that allow computing on encrypted data, specifically executing Procedure PR, we theoretically showed how to read or write data in one round without revealing the type of access.

6.3.3 Challenges with FHE based solution

Although we have shown the theoretical feasibility of using FHE to read or write data obliviously in one round, this approach is not practically feasible, mainly due to the noise n necessary for homomorphic encryption (as shown in Equations 6.2 and 6.3). As noted above, the noise increases with each homomorphic computation, with the increase being especially drastic for homomorphic multiplications, which the Procedure PR requires for both read and write accesses.

To gauge the practicality of the above described FHE based solution, we developed and evaluated a prototype of the solution. The prototype used Microsoft SEAL [155] FHE library with BFV [61] scheme. The evaluation used values of size 1kb and 128-bit secret keys, and BFV coefficients set to their default in the SEAL library.

Our experiments revealed that after about 10 accesses to a specific data item, the noise value grew too large for the FHE decryption to succeed, essentially rendering this solution impractical for any use in real deployments. The inevitable multiplication in Procedure PR for both reads and writes is the root cause of this infeasibility. We believe that our proposed FHE solution can be used in the future if better performing FHE schemes are invented that control the amount of noise amplification.

6.4 ORTOA

Having shown that the use of an existing cryptographic primitive, Fully Homomorphic Encryption (FHE), as-is is impractical to provide the desired one round-trip oblivious access approach, we propose a novel protocol, ORTOA, that avoids FHE.

Since the existing encryption scheme, FHE, failed to provide the desired result, we take a step further and define a rather unique way of encoding the data values stored at the external server. We first consider the plaintext value in its binary format. For each binary bit of the plaintext, the server stores a secret label generated by the proxy using pseudorandom functions. This idea of encoding bits using secret labels is inspired by garbled circuit constructions [217, 132].

More precisely, if k is a data item's key and v its plaintext value in binary, then the server stores:

$$< PRF(k), (sl_{b1}^{(1)}, \ldots, sl_{bj}^{(j)}, \ldots, sl_{b\ell}^{(\ell)}) >$$

where $\ell = |v|$, $sl_{bj}^{(j)}$ is a secret label corresponding to the jth index of v from the left (indicated as the superscript) where j goes from 1 to ℓ, and $\forall j,\ bj \in \{0, 1\}$ represents bit value 0 or 1 (indicated as the subscript). For example if $\ell = 3$ and $v = 101$ (in binary notation) , then the server stores $(sl_1^{(1)}, sl_0^{(2)}, sl_1^{(3)})$. The secret labels are generated using a pseudorandom function of the form $PRF(k, j, b, ct)$ that takes the key k, position index j from left, the corresponding bit value b and an access counter ct. Because PRFs are deterministic functions, invoking the chosen PRF with the same set of inputs any number of times will result in the same output secret label.

The goal of ORTOA is to read and write data in one round-trip, without revealing the type of access. Intuitively, it becomes evident that to hide reads from writes, every access to a data item must write the data, which is what ORTOA does at a high level: it updates the secret labels of a data item whenever a client accesses the item – be it

for a read or a write. We use the notation *ol* to represent the *old* secret label currently stored at the server and *nl* to represent the *new* label that would replace the old label. To be able to regenerate the last array of secret labels for a given data item, the proxy maintains an access counter indicating the total access count of an item.

6.4.1 An Illustrative Example

For ease of exposition, we first explain how ORTOA executes reads and writes using a simple example. We formally present the protocol in the next section.

Recall that all data values are of the same length, ℓ bits, indexed 1 to ℓ. In this example, let $\ell = 1$, and let k be the specific key accessed by a client where the corresponding key-value tuple is $< k, 0 >$, i.e., the value associated with k is 0. Since the server does not store keys or values in plain, the server stores the corresponding tuple $< PRF(k)$, $ol_0^{(1)} >$ where $ol_0^{(1)}$ is a secret label for bit value 0 (indicated as the subscript) at index 1 (indicated as the superscript).

1. Client: The client either sends a `Req(Read, k)` or a `Req(Write, k, v')` request to the proxy, where v' is an updated value for k. In this example, we assume v' is 1.

2. Proxy: The proxy, in response, executes the following steps:

2.1 The proxy generates two ***old*** secret labels $< ol_0^{(1)}, ol_1^{(1)} >$ (where *ol* indicates old label) both for index 1 by calling $PRF(k, 1, b, ct)$ where $b \in \{0, 1\}$ and ct is k's access counter. For each index, the proxy needs to generate labels for both bit values 0 and 1 *since it does not know the actual value, which is stored only at the server*.

2.2 The proxy next generates two ***new*** labels $< nl_0^{(1)}, nl_1^{(1)} >$ (where *nl* indicates new label) both for index 1 by calling $PRF(k, 1, b, ct+1)$ where $b \in \{0, 1\}$ and it updates k's access count to $ct + 1$.

2.3 The details of this step depend on the type of access: for reads, the proxy encrypts each new secret label using the corresponding old secret label, thus generating two encryptions for index 1:

$E = [< Enc_{ol_0^{(1)}}(nl_0^{(1)}), Enc_{ol_1^{(1)}}(nl_1^{(1)}) >]$

Whereas for writes, assuming the updated value $v' = 1$, the proxy encrypts only the new label corresponding to the updated value $v' = 1$ using the old labels, i.e.:

$E = [< Enc_{ol_0^{(1)}}(\boldsymbol{nl_1^{(1)}}), Enc_{ol_1^{(1)}}(\boldsymbol{nl_1^{(1)}}) >]$

2.4 The proxy next shuffles E pairwise, i.e, randomly reorders the two encryptions, to ensure that the first encryption does not always refer to bit 0 and the second to bit 1, and sends E to the external server.

3. Server: The external server, upon receiving E does the following:

3.1 For the pair of encryptions received, the server tries to decrypt both encryptions using its locally stored label. But since it stores only one old label at index 1, it succeeds in decrypting only one of the two encryptions. In this example, the server decrypts $Enc_{ol_0^{(1)}}(nl_0^{(1)})$ for reads or $Enc_{ol_0^{(1)}}(nl_1^{(1)})$ for writes using the stored $ol_0^{(1)}$.

3.2 The server then updates index 1's secret label to the newly decrypted value, in this case, $nl_0^{(1)}$ for reads or $nl_1^{(1)}$ for writes. For writes, since both encryptions for an index encrypt only one new label $nl_1^{(1)}$, either decryptions will result in the desired, updated label that reflects the new value of $< k, 1 >$. Whereas for reads, the server ends up with $nl_0^{(1)}$, reflecting the existing value of $< k, 0 >$. The server sends the output of the decryption to the proxy and since the proxy knows the mapping of secret labels to plaintext bit values, the proxy learns the value of k to be 0 for reads and ignores the output for writes.

Algorithm 8 Procedure Init(kv):

1: $kv' \leftarrow \emptyset$
2: $ct \leftarrow 1$ *// indicates an access count of 1*
3: **for** $(k, v) \in kv$
4: $\quad labels \leftarrow \emptyset$
5: $\quad i \leftarrow 1$ *// index*
6: $\quad$**for** each bit $b \in v$ starting from left most position *// v is in binary representation*
7: $\quad\quad l \leftarrow PRF(k, i, b, ct)$
8: $\quad\quad labels \stackrel{\cup}{\leftarrow} l$
9: $\quad\quad i \leftarrow i + 1$
10: $\quad kv' \stackrel{\cup}{\leftarrow} \{PRF(k), labels\}$
11: Return kv'

Figure 6.1: ORTOA's algorithm for initializing a given set plaintext key value pairs kv.

6.4.2 Protocol

Having described a simple example that uses ORTOA to read and write data without revealing the type of operation, this section formally presents the ORTOA protocol described in the two functions depicted in Figures 6.1 and 6.2. Table 6.1 defines the variables used in explaining ORTOA.

The Init(kv) procedure describes the data initialization process in ORTOA. The procedure receives the key-value pairs in plain text as input. For each key-value pair (line 3), the procedure generates PRF labels at each of the ℓ indexes corresponding to bit b of the value (represented in binary form) (line 7). All the labels appended together represent the value (line 10) and the procedure returns the encoded keys and labels to be stored at the external server.

When a client sends `Req(Read,`k`)` or a `Req(Write,`k`,`v'`)` to the proxy, the proxy and the server execute the following steps.

1. Proxy: The proxy, upon receiving a `Req(Read,` k`)` or a `Req(Write,` k`,` v'`)` request from a client, where v' is an updated value for k, invokes the ProcessClientRequest

Algorithm 9 Procedure ProcessClientRequest(op, k, val)

1: Retrieve key k's ct *// k's latest access count*
2: $E \leftarrow \emptyset$
3: $i \leftarrow 1$ *//index*
4: **for** each bit $b \in val$ starting from left most position *// val is in binary representation*
5: $ol_0^{(i)} \leftarrow PRF(k, i, 0, ct)$, $ol_1^{(i)} \leftarrow PRF(k, i, 1, ct)$
6: $nl_0^{(i)} \leftarrow PRF(k, i, 0, ct + 1)$, $nl_1^{(i)} \leftarrow PRF(k, i, 1, ct + 1)$
7: **if** $op = read$
8: $E \xleftarrow{\cup} \{Enc_{ol_0^{(i)}}(nl_0^{(i)}),\ Enc_{ol_1^{(i)}}(nl_1^{(i)})\}$
9: **else**
10: $E \xleftarrow{\cup} \{Enc_{ol_0^{(i)}}(nl_{b_i}^{(i)}),\ Enc_{ol_1^{(i)}}(nl_{b_i}^{(i)})\}$
11: $i \leftarrow i + 1$
12: $ct \leftarrow ct + 1$
13: Return E

Figure 6.2: ORTOA's algorithm for processing of an individual client request for operation type op, key k, and updated value val.

Symbol	Meaning
$ol_{bj}^{(j)}$	Secret label of a single bit of plaintext value
j	Index from 1 to ℓ starting from the left of plaintext value
bj	Bit value (0 or 1) at index j of plaintext value
ct	Access counter
$nl_{bj}^{(j)}$	New secret label of a single bit of plaintext value

Table 6.1: Variables used in ORTOA.

procedure as defined in Figure 6.2, which internally executes the following steps:

1.1 The proxy retrieves key k's access counter ct (line 1).

1.2 For each of the ℓ indexes of the value, the proxy generates the two *old* secret labels corresponding to both bit-values 0 and 1 by passing the current access counter ct to the PRF (line 5):

$\{ol_0^{(1)} \leftarrow PRF(k, 1, 0, ct),\ ol_1^{(1)} \leftarrow PRF(k, 1, 1, ct),\ \ldots,$

$ol_0^{(\ell)} \leftarrow PRF(k, \ell, 0, ct),\ ol_1^{(\ell)} \leftarrow PRF(k, \ell, 1, ct)\}$

1.3 For each of the ℓ indexes of the value, the proxy next generates two *new* secret labels corresponding to both bit-values 0 and 1 by passing the updated access counter $ct + 1$ (accounting for the ongoing access) to the PRF (line 6):

$\{nl_0^{(1)} \leftarrow PRF(k, 1, 0, ct + 1),\ nl_1^{(1)} \leftarrow PRF(k, 1, 1, ct + 1), \ldots,$

$nl_0^{(\ell)} \leftarrow PRF(k, \ell, 0, ct + 1),\ nl_1^{(\ell)} \leftarrow PRF(k, \ell, 1, ct + 1)\}$

1.4 The details of this step depend on the type of access: for reads, the proxy encrypts each new secret label using the corresponding old secret label and generates two encryptions for each of the ℓ indexes (line 8):

$$E = [< Enc_{ol_0^{(1)}}(nl_0^{(1)}),\ Enc_{ol_1^{(1)}}(nl_1^{(1)}) >, \ldots, < Enc_{ol_0^{(\ell)}}(nl_0^{(\ell)}),\ Enc_{ol_1^{(\ell)}}(nl_1^{(\ell)}) >]$$

Whereas for writes, assuming b_i represents the updated bit value at index i, the proxy encrypts only the new labels corresponding to the updated value v' using the old labels (line 10):

$$E = [< Enc_{ol_0^{(1)}}(\boldsymbol{nl_{b_1}^{(1)}}),\ Enc_{ol_1^{(1)}}(\boldsymbol{nl_{b_1}^{(1)}}) >, \ldots, < Enc_{ol_0^{(\ell)}}(\boldsymbol{nl_{b_\ell}^{(\ell)}}),\ Enc_{ol_1^{(\ell)}}(\boldsymbol{nl_{b_\ell}^{(\ell)}}) >]$$

As noted above for writes, at each index i, both the old labels encrypt only one new label $nl_{b_i}^{(i)}$ corresponding to v'.

1.5 The proxy increments k's access counter (line 12).

1.6 The proxy pairwise shuffles each of the ℓ pairs of encryptions at each index and sends this encryption to the external server.

2. Server: The server upon receiving the encryption E from the proxy performs the following steps:

2.1 For each of the ℓ pairwise encryptions, the server tries to decrypt both encryptions using the locally stored label. However, since it stores only one old label per index, it succeeds in decrypting only one of the two encryptions per index. At index j, the server either stores $ol_0^{(j)}$ or $ol_1^{(j)}$, and hence, it can successfully decrypt only one of $< Enc_{ol_0^{(j)}}(nl_0^{(j)}), Enc_{ol_1^{(j)}}(nl_1^{(j)}) >$ obtaining $nl_0^{(j)}$ or $nl_1^{(j)}$ for reads. Note that for writes, since both encryptions encrypt $nl_{b_j}^{(j)}$, either decryptions will result in the new label corresponding to the updated bit b_j at index j.

2.2 The server then updates each index's secret label to the newly decrypted value and sends the output to the proxy. Since the proxy knows the mapping of secret labels to plaintext bit values at each index, the proxy learns the value of k for reads and it ignores the output for writes.

After executing ORTOA to access a data item the server always updates its stored secret labels. For reads, the updated labels reflect the *existing value* of the data item; for writes, the updated labels reflect the *updated value* of the data item. Thus by choosing a unique data representation model and taking advantage of that model, ORTOA provides a one round-trip oblivious access protocol without restricting the number of accesses, unlike the FHE approach.

6.4.3 Complexity Analysis

Space Analysis

Proxy: The only information the proxy needs to maintain to support ORTOA is the access counter for each key in the database. While the complexity of storing access counters for all the keys is *O(N)*, where N is the database size, the actual space it consumes is quite low. For example if a single counter consumes 8 bytes, for a database of size 1 million items, the proxy requires about 8mB space to store the counters.

Server: While the storage cost at the proxy is insignificant to support ORTOA, the same is not true for the server. The exact space analysis at the server is as follows: if ℓ represents the length of a plaintext value (and all values have same length), r the output size (in bits) of the PRF that generates secret labels, and N the database size, then server's storage space in bits can be calculated as:

$$\underbrace{(r \cdot N)}_{Space\,for\,keys} + \underbrace{(r \cdot \ell \cdot N)}_{Space\,for\,values}$$

Communication Analysis

Every bit of the plaintext can have 2 possible values – either a 0 or a 1. Since the data values, or rather the data value mappings, are stored only at the server, the proxy generates both possible secret labels, and the corresponding 2 encryptions, for each bit of the plaintext. The proxy then sends 2 encryptions per bit to the server. If ℓ be the length of data values and E_{len} the length of encrypted ciphertexts, for every data item

A few plaintext bit combinations	1-label-per-bit representation
0000	$sl_0^{(1)}, sl_0^{(2)}, sl_0^{(3)}, sl_0^{(4)}$
0001	$sl_0^{(1)}, sl_0^{(2)}, sl_0^{(3)}, sl_1^{(4)}$
0010	$sl_0^{(1)}, sl_0^{(2)}, sl_1^{(3)}, sl_0^{(4)}$
0011	$sl_0^{(1)}, sl_0^{(2)}, sl_1^{(3)}, sl_1^{(4)}$

Table 6.2: When $\ell = 4$ and each secret label represents one bit of plaintext data, i.e, $y = 1$.

A few plaintext bit combinations	1-label-per-2-bits representation
0000	$sl_{00}^{(1,2)}, sl_{00}^{(3,4)}$
0001	$sl_{00}^{(1,2)}, sl_{01}^{(3,4)}$
0010	$sl_{00}^{(1,2)}, sl_{10}^{(3,4)}$
0011	$sl_{00}^{(1,2)}, sl_{11}^{(3,4)}$

Table 6.3: When $\ell = 4$ and each secret label represents two bits of plaintext data, i.e, $y = 2$.

accessed by a client, ORTOA incurs the communication cost of:

$$\underbrace{2 \cdot E_{len}}_{Encryptions\ per\ bit} \cdot \underbrace{\ell}_{Number\ of\ bits}$$

6.5 Optimizations

6.5.1 Space optimized solution

In this section, we discuss a technique to optimize storage space by trading off communication cost. Recall that for every bit of plaintext data, the server stores a secret label of r bits; in other words, r bits are used to represent a single bit of plaintext data. To optimize space, the next logical question we ask is: can we use r bits to represent multiple bits of plaintext data?

One label represents two bits of the plaintext: We start with a simple case

where a single label represents two bits of plaintext data (Table 6.3), instead of one (Table 6.2). In this case, the server stores $\ell/2$ labels for every data item (instead of ℓ), reducing the storage space by half. For example, if the plaintext value is 0010, then the server stores $[sl_{00}^{(1,2)}, sl_{10}^{(3,4)}]$ where, say label $sl_{10}^{(3,4)}$ corresponds to plaintext values 1 and 0 at indexes 3 and 4 respectively.

There are $2^2 = 4$ unique bit combinations for every 2 indexes of the plaintext – 00, 01, 10, and 11. Since the proxy does not know the value, which is stored only at the server, it generates 4 secret labels for every 2-bits, i.e., labels for all possible unique bit combinations, and creates 4 corresponding encryptions for every two bits of plaintext data. The proxy then sends these 4 encryptions per 2-bits to the server, which then tries to decrypt all 4 encryptions. Since the server stores only one label per 2-bits, it succeeds in decrypting only one of the 4 encryptions per 2-bits, which becomes the new label for those 2-bits.

One label represents y bits of the plaintext: This approach can be further generalized where a single label represents y bits of the plaintext. For example a label $sl_{b_1 \ldots b_y}^{(1,\ldots,y)}$ corresponds to bits $b_1 \ldots b_y$ from indexes 1 to y. This approach reduces the storage space to ℓ/y, i.e, the storage space reduces by a factor of y. Note that if the length of values, ℓ, is not divisible by y, we can pad the plaintext with a specific character to indicate the bit value at that index is invalid.

Communication complexity increase: While the space optimized solution reduces the storage space at the server by a factor of y, it incurs increased communication and computation overhead as more labels need to be communicated from the proxy to the server, as analysed next. As discussed in §6.4.3, the communication complexity of the non-space-optimized solution is $(2 \cdot E_{len} \cdot \ell)$. Generalising this to when one secret label represents y bits, there are 2^y possible unique combinations for every y bits of plaintext and the server stores ℓ/y labels. So the communication (and computation) complexity

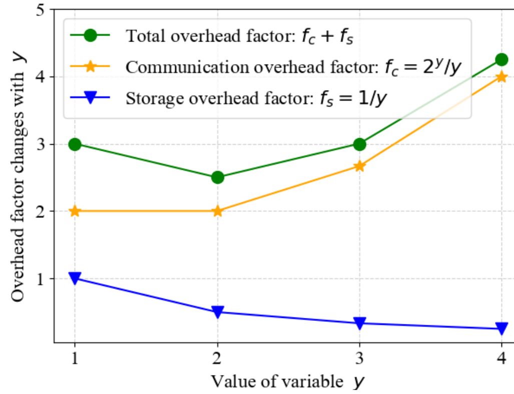

Figure 6.3: Storage vs communication overhead factor analysis to find optimal y value - the value that indicates how many bits are represented by a single label.

becomes $(2^y \cdot E_{len} \cdot \ell/y)$, i.e, a factor of $2^y/y$ increase compared to the non-space-optimized solution.

Calculating optimal y value: The above discussion implies that there exists a trade-off between the storage space and the amount of communication (and computation) with the increase in y. When y increases, the storage space reduces by a factor $f_s = 1/y$ and the communication expense increases by a factor $f_c = 2^y/y$, i.e., while the storage space decreases non-linearly, the amount of communication increases exponentially.

To calculate the optimal value of y, we compare the overhead factors f_s, f_c, and the total combined overhead of the system, $f_s + f_c$, as depicted in Figure 6.3. As expected and as seen in the figure, the storage factor reduces with increasing y, and communication factor increases with y. The total overhead plot is interesting: the overall overhead decreases for $y = 2$ and starts increasing from $y = 3$. This is because when $y = 2$, the storage space reduces by half, meanwhile the communication factor remains the same for $y = 1$ and $y = 2$, with $f_c = 2$. For any $y > 2$, the communication factor increases more rapidly than the storage factor reduction, causing the total overhead factor to increase with y. Since the total overhead is the least at $y = 2$, that becomes the optimal y for ORTOA.

6.5.2 Reducing the number of decryptions

Given that ORTOA has the least overhead for $y = 2$, i,e, a single label representing 2-bits of plaintext, this implies that the proxy sends $2^y = 2^2 = 4$ encryptions for every 2-bits of plaintext. Since the server stores a single label corresponding to a unique bit combination for every 2-bits of plaintext (Table 6.3) , the server can successfully decrypt only one of the 4 encryptions. In the solution presented above, the four encryptions per 2-bits are randomly shuffled by the proxy, and hence, the server attempts to decrypt all encryptions until it succeeds (note that ORTOA uses authenticated encryption to ensure the server identifies successful decryptions). Essentially, the server wastes computation trying to identify the right encryption. To mitigate this inefficiency and reduce the number of potential decryptions on the server from 4 to 1 for every 2-bits of plaintext, ORTOA adapts the point-and-permute [20] optimization.

To reduce the number of decryptions, instead of sending the 4 encryptions per 2-bits in a randomly shuffled manner, the proxy generates the four entries in a deterministic way. For ease of exposition, let us assume that the 4 encryptions are sent as a table where each of the four entries are indexed in binary notation: `00,01,10`, and `11` indicating the 1st, 2nd, 3rd, and 4th entry of the table.

Intuitively, the proxy generates two additional bits of information per label indicating which of the four entries to decrypt upon the next access; we term them ***decryption bits*** d_1d_2. The server stores bits d_1d_2 along with its corresponding secret label. For example, if the server stores a label $(sl_{00}^{(1,2)}, \mathbf{10})$ for the plaintext indexes (1,2) of an item, the decryption bits 10 indicate that the server should decrypt only the $\texttt{10}^{th}$ entry, i.e., the third entry, in the encryption table sent by the proxy for plaintext indexes (1,2). We discuss how the proxy generates the two decryption bits, d_1d_2, next.

To simplify the explanation of the optimization, let us consider $\ell = 2$. The server

stores a single label, $ol_{b_1b_2}$, corresponding to two bits of plaintext of an item, and the decryption bits d_1d_2. The main constraint that the proxy needs to guarantee while generating the encryption table when a client accesses the item next is: the encryption entry at index d_1d_2 uses the label $ol_{b_1b_2}$, i.e., $d_1d_2^{th}$ entry in the table is $Enc_{ol_{b_1b_2}}(nl_{b'_1b'_2})$ where $b'_1b'_2$ is b_1b_2 for reads and the updated bits for writes. This constraint needs to be guaranteed because with this optimization, we are stating that the server decrypts only $d_1d_2^{th}$ entry in the table but the server can only decrypt an encryption that used ol_{b_1b2} (since that is the only label it stores). Essentially, the proxy needs to deterministically 'link' d_1d_2 with b_1b_2 but also randomize this link for every access. The proxy achieves this by leveraging two random bits, r_1r_2, which act as one-time padding bits to link encryption table indexes with labels. Note that the proxy does not store these two bits r_1r_2 explicitly; they can be derived with any PRF (e.g., a PRF $\mathcal{P}$ that takes the access counter ct and key k as input to generate the two bits).

First, let us consider a simplified case where ORTOA supports accessing a data item only once, and hence decryption bits d_1d_2 need not be updated. To access a given item, the proxy generates the four encryption entries for the 2-bits of plaintext by first generating the old and new labels as described in Steps 1.2 and 1.3 of §6.4.2. Next the proxy creates $d_1d_2^{th}$ entry and links it to the labels by xor-ing with bits r_1r_2:
For reads:

$$d_1d_2^{th} entry : Enc_{ol_{d_1d_2 \oplus r_1r_2}}(\, nl_{d_1d_2 \oplus r_1r_2}\,)$$

For writes where $nl_{b'_1b'_2}$ represents the new label (essentially all entries encrypt the same new label as discussed in §6.4.2):

$$d_1d_2^{th} entry : Enc_{ol_{d_1d_2 \oplus r_1r_2}}(\, nl_{b'_1b'_2}\,)$$

To generalize this, where ORTOA supports any number of accesses to an item, the two decryption bits need to be updated after each access. Essentially, at *each* access, we update the decryption bits to $d'_1d'_2$ indicating which entry to decrypt upon the *next* access. The proxy achieves this by generating two new bits r'_1 and r'_2 using the same PRF that generated $r1$ and $r2$ (e.g., invoke PRF $\mathcal{P}$ with updated access counter $ct+1$ and k). The proxy generates the encryption table with four entries as follows:

For reads:

$$d_1d_2^{th}entry : Enc_{ol_{d_1d_2 \oplus r_1r_2}}(\underbrace{nl_{d_1d_2 \oplus r_1r_2}}_{New\ label}, \underbrace{d_1d_2 \oplus r_1r_2 \oplus r'_1r'_2}_{Bits\ d'_1d'_2})$$

For writes where $nl_{b'_1b'_2}$ represents the new label :

$$d_1d_2^{th}entry : Enc_{ol_{d_1d_2 \oplus r_1r_2}}(\underbrace{nl_{b'_1b'_2}}_{New\ label}, \underbrace{d_1d_2 \oplus r_1r_2 \oplus r'_1r'_2}_{Bits\ d'_1d'_2})$$

The server upon receiving the encryption table decrypts one entry based on the decryption bits d_1 and d_2. A decryption yields both the new label as well as the updated bits d'_1 and d'_2, which determines what entry to decrypt for the next access. This approach can be generalized to values of any arbitrary length ℓ. Thus by constructing an optimization similar to point-and-permute technique, ORTOA reduces the potential number of decryptions performed by the server from 4 to 1.

6.6 Protocol evaluation

In this section, we discuss the merits and limitations of ORTOA by conducting experimental evaluations. In evaluating ORTOA, we consider a two-round-trip (2RTT) protocol as the baseline: the baseline system also consists of a proxy, which routes client

requests to the external server. The baseline proxy translates each request by a client – read or write – into a read request followed by a write request. This technique is on par with how majority of the existing obliviousness solutions hide the type of operation [90, 195, 193, 184]. Note that since ORTOA focuses only on hiding the type of operation, the baseline also mimics the same behavior and hence does not hide the physical location being accessed by a client.

Experimental Setup: We evaluated ORTOA and its baseline on AWS. The clients were deployed on a c5.large instance with 8GiB of memory with 2 cores @ 3.6GHz. The proxy was evaluated on a c5.2xlarge instance with 8GiB of memory and 8 cores @ 3.6GHz. The server was evaluated on an r5.xlarge instance with 8GiB of memory and 4 cores @ 3.1GHz. The client and proxy were located in the US-West1 (California) datacenter and in most of our experiments, the server was hosted in the US-West2 (Oregon) datacenter.

Unless stated otherwise, in each experiment a multi-threaded client (with a default of 32 threads) sends requests concurrently to the proxy and the experiment runs until each client thread sends 100 requests. Each thread sends requests sequentially, i.e., it waits until its current request is answered before sending the next one. Each data point plotted in all our experiments is an average of 3 runs. In our experiments, the server stores 10000 data items, which results in 2GB of data in memory. Note that the baseline requires 40.96MB to store 10000 data items. Each client thread picks a key to access uniformly at random, and unless stated otherwise, it decides to read or write the data also uniformly at random. Most of the experiments choose a 1kB value size, $\ell = 8000$ bits. Each experiment measures *latency*, the time interval between when a client sends a request to when it receives the corresponding response; and *throughput*, the number of operations executed per one second.

6.6.1 ORTOA vs. two round trip baseline

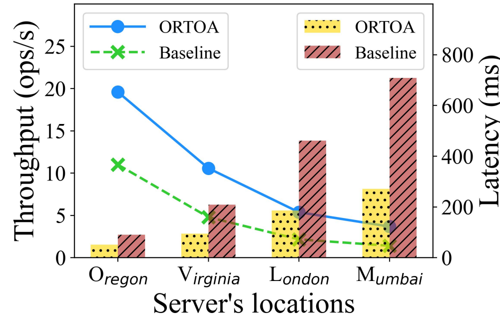

Figure 6.4: Throughput and latency for ORTOA and the 2RTT baseline, with the proxy in the California datacenter and the server placed at increasingly farther datacenters.

In the first set of experiments, we compare ORTOA with the 2RTT baseline where the proxy and client are located in the US-West1 (California) datacenter and the server is placed at increasingly farther datacenters of US-West2 (Oregon), US-East1 (N. Virginia), EU-West2 (London), and AP-South1 (Mumbai). Table 6.4 notes the round-trip time (RTT) latencies from California to the other datacenters. The measured throughput and latency are plotted in Figure 6.4. Note that we do not place the server in the same datacenter as the proxy and client so as to mimic realistic behavior where between 79%-95% of cloud users face more than 10 ms latency when accessing a cloud server [40]. Further, this experiment runs a single-threaded client since our goal is to measure the effect of proxy-to-server distance on a given client's throughput and latency, without accounting for the performance effects due to concurrency.

As seen in Figure 6.4, as the physical distance between the proxy and the server increases, latency increases and throughput decreases for both ORTOA and the 2RTT baseline. But the latency of the 2RTT baseline is **0.76x-1.61x** higher than ORTOA, and its throughput is **43%-61%** lower than ORTOA. This experiment highlights the benefits of constructing a protocol that can hide the type of access in a single round, as compared

to the state-of-the-art two-round approach.

6.6.2 Latency breakdown of ORTOA

	Oregon	N. Virginia	London	Mumbai
California	21.84	62.06	147.73	230.3

Table 6.4: RTT latencies across different datacenters in ms.

	Oregon	N. Virginia	London	Mumbai
Computation (ms)	14.01	14.21	14.42	14.48
Communication (ms)	37.03	80.37	171.78	257.14
Total time (ms)	51.04	94.59	186.21	271.62

Table 6.5: Time spent in computation (creating old and new labels and encrypting them) vs. time spent in communication, in ms, when the proxy and client are located in California and the server is located at different datacenters.

Since ORTOA's computation cost is high due to generating old and new labels for every 2-bits of plaintext and performing 4 encryptions for every 2-bits of data, in this experiment, we measure the time spent by the proxy in computation vs. in communication. Similar to the last experiment, this experiment places the proxy and the client in the US-West1 (California) datacenter and the server at increasingly farther distances from US-West1. Table 6.4 records the round trip time (RTT) from California to the other datacenters and Table 6.5 notes the average computation time vs. communication time and the total time, in milliseconds, spent by ORTOA in executing a request. As shown in Table 6.5, ORTOA consistently spends 14 ms in computing the labels and encrypting the data. In the total time spent per request, ORTOA spends the majority of the time in communication. This latency breakdown also indicates when is ORTOA

a better choice compared to the 2RTT baseline: let c be the round-trip latency between the proxy and the server. If $2 * c < 14$ ms, then this indicates that two sequential rounds of communication requires less time than the computation time of ORTOA, and hence the 2RTT baseline is a better choice for an application choosing between ORTOA and the 2RTT solution. But since most cloud users face over ($c =$) 10 ms latency in accessing a cloud server [40], most applications will save latency by choosing ORTOA.

6.6.3 Increasing Concurrency

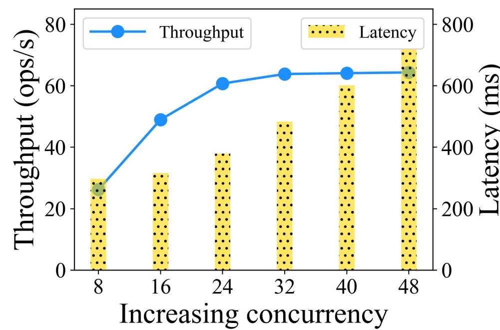

Figure 6.5: ORTOA's throughput measured with increasing the number of concurrent clients.

Having compared ORTOA with its 2RTT baseline, we now evaluate ORTOA's behavior with increasing concurrent client requests. In this experiment, we place the server at the US-West2 (Oregon) datacenter. Figure 6.5 depicts the throughput and latency changes as the number of concurrent clients (implemented via threads) increases from 8 to 48. As seen in the figure, the throughput increases by **1.4x** at 32 clients compared to 8 clients and the throughput saturates at ∼64 ops/sec for higher concurrency values. Since a concurrency of 32 clients has the lowest latency while providing close to peak throughput, the following experiments choose the concurrency of 32 clients.

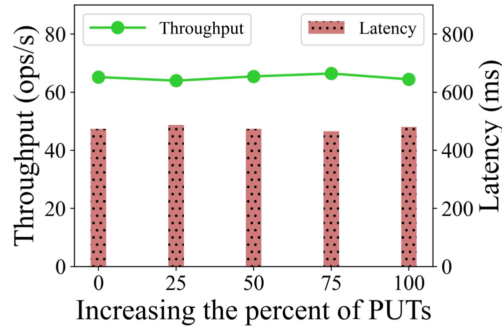

Figure 6.6: ORTOA's throughput measured with increasing percent of PUTs.

6.6.4 Varying the percent of writes

This experiment measures ORTOA's throughput and latency while increasing the percent of PUT (or write) operations from 0 to 100%, as shown in Figure 6.6. In this experiment, the server resides at the US-West2 (Oregon) datacenter and 32 concurrent clients read or write the data. As seen in the figure, the throughput and the latency values remain more or less constant (a maximum difference of 2 ops/sec for throughput and 24 ms for latency). This experimentally demonstrates the obliviousness of ORTOA in that its performance remains the same regardless of the percentage of read or write operations in the client workload.

6.6.5 Varying ℓ: the length of values

Since the storage, communication and computation overhead of ORTOA are directly proportional to ℓ (see §6.4.3), in this experiment, we measure ORTOA's throughput and latency while increasing the size of the values (where all values have the same length) from 200B to 1.2kB (or 1600 to 9600 bits) and the results are depicted in Figure 6.7. The server in this experiment resides in US-West2 (Oregon) datacenter and the client has 32 concurrent threads sending read or write requests. As expected, ORTOA's performance, both in terms of throughput and latency, degrades almost linearly with the increase in

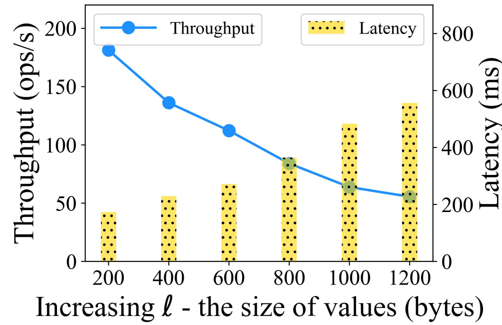

Figure 6.7: ORTOA's throughput and latency measured when the size of the values, ℓ, increases from 200B to 1.2kB (1600 to 9600 bits).

the value size. This experiment indicates that ORTOA suits applications with smaller value sizes rather than with larger value sizes.

6.6.6 Scaling ORTOA

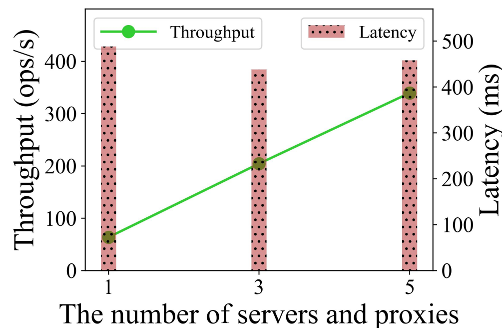

Figure 6.8: ORTOA's throughput and latency measured when the number of servers and proxies in the system are scaled up.

In the final set of experiments, we highlight the scalability of ORTOA by increasing the number of servers and proxies from 1 to 5. Since ORTOA aims to hide only the type of access performed by a client, the system can scale the number of proxies without compromising security. In this experiment, we pair each storage server with a proxy and scale the pair. For each scaling factor s, the client concurrency is also increased by the scaling factor $32 * s$. This experiment places all the proxies and clients in the

US-West1 (California) datacenter and the servers in the US-West2 (Oregon) datacenter. The resulting throughput and latency are shown in Figure 6.8. As indicated in the plot, ORTOA scales linearly with the increasing number of proxies: its peak throughput at a scale factor of 5 is about **5x** the throughput at a scale factor of 1. The latency remains roughly constant (a maximum difference of 50 ms) across different scale factors. This experiment highlights the that ORTOA is highly scalable – a highly desired property of data management protocols.

6.6.7 Discussion

Through the above discussed experimental evaluations of ORTOA, we have shown the benefits of a single round protocol that hides the type of operation. Since ORTOA incurs high storage and communication overheads, in this section, we discuss the estimated dollar cost of deploying ORTOA. To calculate the estimates, we consider the storage, communication, and compute costs of Google Cloud [83, 85], whose costs are comparable to other cloud providers. Google Cloud charges $0.02 per GB of storage per month, $0.12 per GB of network usage and $0.4 per million function invocations with a 1.4 GHz CPU costing $0.00000330 per 200ms (ORTOA needs 145 ms to encrypt/decrypt a label). In estimating the dollar cost, we consider the optimized protocol with $y = 2$, and PRFs that produce 128-bit labels, i.e., $r = 128$, with data values of size 1kB, i.e., $\ell = 8000$, and with encryption schemes that produce 128-bit ciphertexts, i.e., $E_{len} = 128$. With the above configuration, consider running ORTOA with a large dataset consisting of 1 million data items. This costs an application **$1.28** in storage per month, and executing 1 million accesses will cost **$30.72** in terms of bandwidth and **$7.00** in terms of compute (function calls). Taking into account the cost of a single access, ORTOA incurs a cost of **$0.000038** per request – a comparable price given that the 2RTT baseline incurs 0.76x-

1.61x higher latency overhead and decreases the overall request serving rate by 43%-61% compared to ORTOA.

6.7 Security of ORTOA

This section defines and proves the security guarantees of ORTOA. ORTOA aims to hide the type of client access – read or write – from an adversary that controls the external database server. To capture this read or write obliviousness, we introduce a security definition called real-vs-random read-write indistinguishability or `ROR-RW` indistinguishability.

Algorithm 10 Security game where given a sequence of client generated accesses A, the Real world takes A as input and the Ideal world takes the sequence of keys accessed in A as input and both produce a sequence of encryptions that are sent to the external server as output.

1: Real(A)

2: $output \leftarrow \emptyset$

3: **for** $a_i \in A$

4: $\quad output \overset{\cup}{\leftarrow} Process - ClientRequest(a_i)$

5: Return *output*

1: Ideal(K)

2: $output \leftarrow \emptyset$

3: **for** $k_i \in K$

4: $\quad output \overset{\cup}{\leftarrow} Simulator(k_i)$

5: Return *output*

Algorithm 11 The Simulator's pseudocode accessed in the the Ideal algorithm.

1: Procedure Simulator(k)
2: $E \leftarrow \emptyset$
3: *//Iterate over each of the ℓ indexes*
4: **for** ($i = 0; i < \ell; i++$)
5: Retrieve the old label $ol^{(i)}$ for k
6: $nl^{(i)} \xleftarrow{\$} \{0,1\}^{\lambda}$
7: $ol'^{(i)} \xleftarrow{\$} \{0,1\}^{\lambda}$
8: $E \xleftarrow{\cup} \{Enc_{ol^{(i)}}(nl^{(i)}),\ Enc_{ol'^{(i)}}(0)\}$
9: $ol^{(i)} \leftarrow nl^{(i)}$
10: Return E

Security definition: Consider a sequence of m client accesses

$$A = \{(op_1, k_1, val_1), \cdots, (op_i, k_i, val_i), \cdots, (op_m, k_m, val_m)\}$$

where for i^{th} request, op_i indicates the type of operation (read or write), k_i denotes the key, and val_i is either an updated value for writes or $\perp$ for reads. In our security definition, the sequence of accesses A is given as input to both the real system and an ideal system (simulator based), where both are stateful entities, and both produce outputs Out_{Real} and Out_{Sim} respectively consisting of a sequence of accesses to the external server. A system is said to be `ROR-RW` secure if, given the two outputs, an adversary can distinguish between the two with negligible probability, i.e.,

For all probabilistic polynomial adversaries $\mathcal{A}$,

$$\mid Pr[A(Out_{Real}) \rightarrow 1] - Pr[A(Out_{Sim}) \rightarrow 1] \mid \leq negl$$

To argue for ORTOA's correctness, we consider a game $\mathcal{G}$, as shown in Algorithm 10. In proving ORTOA's correctness, we assume the length ℓ of data values to be 1 but the argument can be generalized to data values of any arbitrary length. Further, our proof considers the non-optimized protocol as presented in §6.4.2 but the proof easily extends to the optimized versions as well. The game either executes Real or Ideal algorithm with uniformly random probability and provides the output to an adversary. ORTOA is `ROR-RW` secure if the adversary, based on the received output, cannot distinguish with high probability which system the game selected.

For Real algorithm in Algorithm 10, the game sends a sequence of m accesses in A produced by clients where the algorithm in-turn calls ORTOA's *ProcessClientRequest* procedure (defined in Figure 6.2) for each access in A. Note that the *ProcessClientRequest* procedure is a stateful algorithm. Let λ be the length of old and new labels and let *Enc* be the encryption scheme deployed in the *ProcessClientRequest* procedure that encrypts new labels of length λ using old labels of length λ. Since we assume $\ell = 1$, *ProcessClientRequest* produces two encryptions for each of the accesses (in real deployments, the proxy sends each of these set of encryptions to the external server upon each access). The Real algorithm collates the output consisting of a pair of encryptions produced by each call to *ProcessClientRequest* method and produces the output. The Real algorithm's output can be represented as:

$$Out_{Real} \leftarrow \{Enc_{ol_b}(nl_{b'}), Enc_{ol_{1-b}}(nl_{b''})\}^m$$

where for each read accesses ($b' = b$) and ($b'' = 1-b$), and for write accesses ($b' = b'' = \hat{b}$), the updated bit.

For the Ideal algorithm in Algorithm 10, the game provides the sequence of keys accessed in A as input where the algorithm in-turn calls a Simulator defined in Algorithm 11.

The Simulator's goal is to produce encryptions similar to the *ProcessClientRequest* procedure but with arbitrary values; one can notice the analogies between the two procedures. To achieve this, we assume the Simulator to be stateful and it stores one old label ol per index i of the value of a given key k – these are the labels stored at the external server. The procedure takes key k as input and iterates over each of the ℓ indexes (where ℓ is the value's plaintext length). At each index, the Simulator retrieves the corresponding old label; it then generates two randomly sampled labels $nl^{(i)}$ and $ol'^{(i)}$ of length λ (same as the PRF used in *ProcessClientRequest*). It uses $ol^{(i)}$ to encrypt $nl^{(i)}$ and $ol'^{(i)}$ to encrypt an invalid value, 0. This does not reveal any information to the adversary that controls the external server since the server only stores label $ol^{(i)}$ and can decrypt only one of the two encryptions sent by the Simulator. The Simulator shuffles the two encryptions at each index and appends it a list E sent to the server. It also updates the old labels $ol^{(i)}$ with the newly and randomly generated label $nl^{(i)}$. Because the Simulator encrypts random values of length λ, the Ideal algorithm's output is, assuming $\ell = 1$:

$$Out_{Sim} \leftarrow \{Enc_{\{0,1\}^\lambda}\{0,1\}^\lambda, Enc_{\{0,1\}^\lambda}\{0,1\}^\lambda\}^m$$

If we can prove that the output generated by the Real algorithm appears indistinguishable to Out_{Sim}, it proves that ORTOA is `ROR-RW` secure.

Proof intuition: Intuitively, we first show that a read and a write access to *ProcessClientRequest* procedure are indistinguishable, and we then show that *ProcessClientRequest*'s output is indistinguishable from that of the Simulator. Algorithm 12 captures the argument for this indistinguishability. The basis of our argument lies in the PRF deployed in ORTOA: ORTOA's PRF, PRF, produces labels that are indistinguishable from a uniformly sampled random variable $r \xleftarrow{\$} \{0,1\}^\lambda$. The argument invokes *ProcessClientRequest* procedure once to read a key k and once to write a key k with

the updated bit value b' (assuming the length of the value $\ell = 1$). As shown in the figure, given that the server stores only one old label ol_b and given PRF's security, the output produced by both invocations of $ProcessClientRequest$ are identical.

When the Real algorithm invokes $ProcessClientRequest$ m times (for m accesses in A), the output of the Real algorithm based on the argument shown in Algorithm 12 becomes indistinguishable from that of Out_{Sim}. We utilize this intuition in developing the formal security proof using hybrids.

Algorithm 12 Intuition for read-write indistinguishability when a key k is accessed where the server stores label ol_b corresponding to k's plaintext value $b \in \{1, 0\}$. The write request updates k's value to bit b'. The PRF deployed in ORTOA generates labels of length λ.

1: For Read Requests
2: $\{Enc_{ol_b}(nl_{b'}),\ Enc_{ol_{1-b}}(nl_{1-b'})\} \leftarrow ProcessClientRequest(read, k, \perp)$
3: *//Because the server has only ol_b, it cannot decrypt $Enc_{ol_{1-b}}(nl_{1-b'})$. So it can be replaced with a random string.*
4: $\equiv \{Enc_{ol_b}(nl_{b'}),\ Enc_{ol_{1-b}}(\{0,1\}^{\lambda})\}$
5: *// From PRF's security, the new label can be replaced with a random string of length λ.*
6: $\equiv \{Enc_{ol_b}(\{0,1\}^{\lambda}),\ Enc_{ol_{1-b}}(\{0,1\}^{\lambda})\}$
7: *// From PRF's security, the old labels can be replaced with random strings of length λ.*
8: $\equiv \{Enc_{\{0,1\}^{\lambda}}(\{0,1\}^{\lambda}),\ Enc_{\{0,1\}^{\lambda}}(\{0,1\}^{\lambda})\}$

1: For Write Requests
2: $\{Enc_{ol_b}(nl_{b'}),\ Enc_{ol_{1-b}}(nl_{b'})\} \leftarrow ProcessClientRequest(write, k, b')$
3: *Because the server has only ol_b, it cannot decrypt $Enc_{ol_{1-b}}(nl_{b'})$. So it can be replaced with a random string.*
4: $\equiv \{Enc_{ol_b}(nl_{b'}),\ Enc_{ol_{1-b}}(\{0,1\}^{\lambda})\}$
5: *// From PRF's security, the label can be replaced with a random string of length λ.*
6: $\equiv \{Enc_{ol_b}(\{0,1\}^{\lambda}),\ Enc_{ol_{1-b}}(\{0,1\}^{\lambda})\}$
7: *// From PRF's security, the old labels can be replaced with random strings of length λ.*
8: $\equiv \{Enc_{\{0,1\}^{\lambda}}(\{0,1\}^{\lambda}),\ Enc_{\{0,1\}^{\lambda}}(\{0,1\}^{\lambda})\}$

Formal proof: We now formally prove that the real and the ideal worlds are com-

putationally indistinguishable using a standard hybrid argument.

Hybrid_1: This corresponds to the real experiment and the output of this hybrid is Out_{Real}.

Hybrid_2: We modify the real experiment where the labels generated using PRF in the *ProcessClientRequest* procedure are now sampled from the uniform distribution.

The computational indistinguishability of Hybrid_1 and Hybrid_2 follows from the security of PRF.

$\mathsf{Hybrid}_{3.i}$ for $i \in [m]$: In the sequence of m accesses in A, consider the ith access, in which the *ProcessClientRequest* procedure generates $2 * \ell = 2 * 1 = 2$ encryptions ($\ell = 1$). Since the server stores only one label per index and can only decrypt one of the two encryptions, the other encryption sent has no significance: let the two ciphertexts be CT_0 and CT_1 where both the ciphertexts are encrypted with respect to two different old labels ol_0 and ol_1. Note that the server has exactly one label ol_b for some bit b. Replace the message in CT_{1-b} with 0s - this encryption becomes *insignificant* since the server cannot decrypt it. This hybrid replaces encryptions of all such insignificant entries with the encryptions of an invalid value, say 0.

The computational indistinguishability of $\mathsf{Hybrid}_{3.i}$ and $\mathsf{Hybrid}_{3.i-1}$ follows from the security of encryption.

Hybrid_4: This corresponds to the ideal experiment, i.e., Out_{Real} is equivalent to Out_{Sim}.

The hybrids Hybrid_4 and $\mathsf{Hybrid}_{3.m}$ are identically distributed. The transition from $\mathsf{Hybrid}_{3.m}$ to Hybrid_4 is as follows: in $\mathsf{Hybrid}_{3.m}$, the labels are still associated with bits and only one of the two encryptions per index generated using the labels is valid. This implies that only one label per index has significance. But note that in $\mathsf{Hybrid}_{3.m}$, the labels are independent of the bits associated with them (due to Hybrid_2). This essentially leads to the conclusion that irrespective of the type of operation, only one of the two encryption

is valid and the valid encryption encrypts a label generated uniformly at random (new label) using another label generated uniformly at random (old label). This is equivalent to the encryptions generated by the Simulator in the ideal world. Hence, the output of this hybrid corresponds to the output of the simulator, Out_{Sim}.

6.8 Conclusion

In this work, we propose ORTOA, a One Round Trip Oblivious Access protocol that reads or writes data stored on remote storage, potentially controlled by an adversary, in a single round of communication without revealing the type of access. Oblivious access techniques consists of two components: obfuscating the data item accessed by a client and hiding the type of client's access, i.e., read or write. Most existing obliviousness solutions focus on obfuscating the data item accessed by a client; whereas to hide the type of access, they require two rounds of communication. To our knowledge ORTOA is the first generalized protocol to hide the type of access in a single round. Experimentally evaluating ORTOA and comparing it with a a baseline that requires two rounds to hide the type of access confirmed the benefits of designing a single round solution: the baseline incurred **0.76x-1.61x** higher latency and **43%-61%** lower throughput than ORTOA. This work also presents a theoretically sound one round trip oblivious access solution using Fully Homomorphic Encryption and discusses its improbability of practical use due to the expensive multiplication operation. As future work, we aim to integrate ORTOA into an end-to-end system that hides access pattern by integrating it with existing techniques such as frequency smoothing or by designing novel ORAM schemes that leverage ORTOA to access data in a single round.

Chapter 7

Concluding remarks

Individuals and enterprises continue to produce ever increasing amounts of data. Much of this data - including sensitive and private information - is stored with and managed by third parties, such as Amazon Web Services or Google Cloud. These companies can lose millions to billions of dollars in sales if their data access latencies increase by only a few hundred milliseconds. Hence, reducing data access latency to improve performance received the highest priority while designing cloud data management systems. But the ever growing number and sophistication of cyber attacks on the cloud coupled with increases in legal requirements for data privacy and security (e.g., GDPR or HIPAA) have forced cloud providers to re-evaluate their priorities. However, there exists a fundamental trade off between security and efficiency in data management systems. While resolving this tension is challenging, it has fostered the growth of a deep field at the intersection of cryptography and database research. While providing high performance and security forms the more explicitly desired properties of data management systems, we have the implicit requirement of *fault tolerance*. We cannot design database systems that are not fault tolerant. Achieving all three desired goals of *low latency & high performance*, *fault tolerance*, as well as *security & privacy* is an extremely challenging problem. This disser-

tation presents data management systems and protocols that strike a balance between the three desired goals of cloud-based data systems.

Trusted infrastructure

Before being able to solve security challenges in database systems, we first had to fully understand existing system designs. Our extensive study of distributed data management, where data is partitioned and/or replicated across geo-distributed locations, led us to identify open problems in traditional, trusted cloud databases. Data in the cloud is partitioned for scalability and replicated for fault tolerance. Atomic commit protocols such as 2-Phase-Commit provide scalability and consensus protocols such as Paxos achieve replication. Existing cloud data management protocols treat atomic commitment and consensus disjointedly. Our work (VLDB 2019) unifies the two seemingly disparate paradigms into a single framework called Consensus and Commitment(C&C). The C&C framework can model established data management protocols as well as propose new ones; to highlight its advantages, we propose a novel commit protocol, G-PAC. The unified approach of G-PAC reduces one (out of three) round of cross-datacenter communication compared with Google's Spanner; this allows G-PAC to perform between 27-88% better than Spanner in terms of throughput.

After instantiating G-PAC from the C&C framework to commit geo-distributed transactions for generalized workloads, we were interested to tackle the problem of handling extremely skewed workloads (such as hotspots) in geo-distributed settings. In this regard, we developed Samya (ICDE 2021) as a database system that improves latency for high contention, hotspot data compared to existing databases. Samya is a geo-distributed data management system that stores and manages hotspot aggregate data. Samya disaggregates aggregate data and stores partitions of the dis-aggregated data on different

servers. This design choice allows servers in Samya to independently and concurrently serve client requests. State-of-the-art geo-distributed databases such as Google's Spanner take a centralized approach where a server acting as a leader processes client requests sequentially. Compared to the centralized solution, Samya's parallelism reduces the 99th percentile latency by 76% and allows Samya to commit 16x to 18x more requests.

Untrusted infrastructure

While G-PAC and Samya successfully iterated on traditional cloud infrastructure, we became intrigued by the question, "What if we host out data on completely untrusted infrastructure?". When designing data management systems for an application that hosts its data on untrusted infrastructure, we primarily focus of providing security while striving for the best possible performance. With regard to security, the solutions presented in this dissertation focus on guaranteeing the *CIA* triad of security: *confidentiality, integrity,* and *availability.*

Ensuring data integrity: To tackle the problem of a database system that guarantees integrity of outsourced data, this dissertation proposed Fides (ICDCS 2020). Fides allows an application to execute distributed transactions on data stored across completely untrusted servers. Through an audit mechanism, Fides guarantees detecting any violations to the ACID (atomicity, consistency, isolation, and durability) properties. This work also presents TFCommit, an integral part of Fides, which is a trust-free commitment protocol that executes transactions across multiple untrusted servers. To our knowledge, TFCommit is the first atomic commitment protocol to tolerate malicious failures without using replication. TFCommit combines a transaction commitment protocol with collective signatures and produces a tamper-resistant log; this log can then be audited to detect faulty behavior. Compared to executing a transaction in trusted infrastructures, the overhead

of executing TFCommit is 1.8x in latency and 2.1 in throughput - an acceptable overhead given the additional security guarantees of TFCommit.

Ensuring data availability: In the context of secure and private datastores, a prevalent area of research is in designing *oblivious* datastores. Oblivious datastores that use the cryptographic technique of Oblivious RAM (or ORAM) provide strong privacy guarantees beyond encrypting the outsourced data by hiding the access patterns on the data. But almost all existing ORAM datastores are not fault tolerant in that if the external server storing the encrypted data or the trusted proxy maintaining the encryption key (and other meta data) crash, the application's data becomes completely unavailable. To solve the data availability issue in oblivious datastores, this dissertation proposed, *QuORAM*, a quorum based replicated ORAM datastore that tolerates crash failures while preserving obliviousness. QuORAM replicates data on $2f + 1$ server-proxy units to tolerate up to f server or proxy crash failures. Furthermore, QuORAM provides linearizable guarantees, which ensures that all operations on a data item appear to be linear. QuORAM reduces the average data access latency by 61.6% and improves the throughput by 1.4x compared to a non-replicated ORAM system, while providing fault tolerance.

Ensuring data confidentiality: Oblivious datastores achieve two goals: (i). hide the specific data item accessed by a client, and (ii). hide the type of access – read or write – requested by a client. To hide the type of access, oblivious datastores typically execute a two-round procedure where a trusted proxy reads the data from the external storage server, re-encrypts the read value if a client requested to read the data or encrypt the newly updated data if a client requests a write operation, and update the re-encrypted or newly encrypted data back in the storage server. This incurs one additional and unnecessary round of communication compared to non-private datastores. To mitigate this inefficiency, in this dissertation we proposed ORTOA, a One Round Trip Oblivious

Access protocol that hides the type of operation in a single round. This reduction in one round of communication plays a vital role in reducing end-to-end latency, especially in geo-distributed settings. Our experimental evaluations show that compared to ORTOA a baseline that requires two rounds to hide the type of access incurs **0.76x-1.61x** higher latency and 43%-61% lower throughput than ORTOA.

In conclusion, this dissertation proposes a number of novel data management systems and protocols that enhance one or more of the desired data system properties of *low latency & high performance*, *fault tolerance*, and *security & privacy*. The solutions proposed perform order of magnitude better than their state of the art counterparts or tackle completely open and unsolved problems such as atomic commitment in malicious settings or fault tolerance via replication in oblivious datastores.

Chapter 8

Future directions

This chapter discusses future research directions that can stem from the protocols and systems proposed in this dissertation.

8.1 Frequency Smoothing using a BST Framework

As stated in the earlier chapters, researchers have exploited the access patterns alone on encrypted data to reveal non-trivial information about a system [105, 91, 110, 114, 34, 55]. While Oblivious RAM, or ORAM, [78, 194, 192, 193, 23, 184, 136, 47, 36] technique mitigates access pattern attacks, ORAM schemes incur fundamental performance overheads [28, 129, 130, 175, 178, 214, 33]. Recently, Grubbs et al. proposed Pancake [90] as a key-value store that hides access patterns with constant storage and bandwidth overheads. At its core, Pancake introduces *frequency smoothing* idea where a trusted proxy smoothens access frequencies of all items stored on the external server such that an adversary cannot infer any information based on clients' real access frequencies. Pancake uses a weaker-than-ORAM yet practical security model of *passive persistent adversary* that can view online queries but cannot inject queries (for example via compromised

clients). To guarantee obliviousness, Pancake requires a priori knowledge on the real access frequencies of all data items in the system. If Pancake estimates an incorrect real access frequency, its security guarantees or performance benefits can be compromised.

To mitigate challenges pertaining to knowing access frequencies a priori, we aim to propose Waffle – an oblivious system that achieves frequency smoothing without relying on an estimated real access frequency. To achieve this, we introduce the Binary Search Tree (BST) Framework that Waffle relies on for frequency smoothing. A trusted proxy in Waffle maintains a balanced BST that stores a key's *access frequencies*, i.e., the number of times an item, identified by its key, stored on the external server is accessed. Any time the proxy accesses a key on the storage server – either for a read or a write operation – the proxy increments the key's frequency and re-balances the tree based on the key frequencies. Waffle utilizes the BST to ensure that the maximum and minimum access frequencies of any two items in the system do not differ beyond a set threshold, Δ.

The key benefit of leveraging a binary search tree to maintain frequency is that it helps track the frequency difference between the least accessed and most accessed item with $\mathsf{O(1)}$ complexity (by maintaining pointers to the left-most and right-most BST nodes). And the properties of a balanced BST allows the proxy to update frequencies and re-balance the tree in $\mathsf{O(logN)}$ complexity where N is the number of data items.

Unlike Pancake [90] where the proxy needs to know the frequency distribution of all keys *a priori*, Waffle's choice of utilizing the BST Framework to dynamically maintain access frequencies removes the stringent requirement of a-priori frequency distribution knowledge. Some key advantages of using the framework to hide the frequencies are as follows:

- No requirement on knowing frequency distribution ahead of time.

- Automatically handles any changes in the frequency distribution.

- Easy to ensure the least and most accessed frequencies do not digress.
- Ease of picking keys for a fake access necessary to hide the real access frequencies.

8.2 High functionality oblivious datastores

Database researchers, and people in computing in general, are grappling with the increasing challenge of data privacy. Laws such as GDPR and HIPAA make data privacy not merely a desired property but a necessary property that database systems must provide. Merely encrypted data is susceptible to privacy attacks based on access patterns, and Oblivious RAM or ORAM, a cryptographic technique, protects data from access pattern attacks. Although traditional trust-assuming database systems have evolved significantly over the years to provide a rich set of features such as concurrency control, transactional ACID guarantees, and query optimizations, almost all currently existing privacy preserving oblivious datastores strip away these features and downgrade the untrusted backend server to a simple system that supports single item Gets and Puts. This simplifying of the backend database servers is usually necessary to uphold the strict definitions of obliviousness, as different database features may reveal non-trivial information about the data. For example: allowing the untrusted database server to handle concurrency control may reveal high contention in application workloads, as multiple concurrent requests might be accessing the same data item. Today, a trusted proxy server provides much of these features such as concurrency control and the backend server merely Gets and Puts individual data items.

We believe an important future research direction will be in developing feature-rich databases, where the features themselves preserve privacy, and to remove the trusted proxy, allowing the clients to access the database servers directly. To achieve this, we aim to consider each database feature separately and build oblivious versions of those features.

For example, concurrency control can be made oblivious by creating fake accesses to data items and mixing them with real access such that from the server's perspective, all data items have equal concurrency. Such a feature-by-feature approach to integrate obliviousness into database systems can help bridge the gaps between theoretical oblivious datastore constructions and existing practical databases.

With respect to building a transactional oblivious datastore, while the computing literature consists of many ORAM solutions, to this day, Obladi [47] by Crooks et al. is the only ORAM solution that provides transactional guarantees. But even Obladi does not address the problem of obliviously executing distributed transactions – transactions that access data stored across different servers. We believe that a future research project can design a distributed database system that guarantees oblivious serializable transactions. One of the main challenges in ORAM systems today is that once a (uncommitted) write operation propagates to the server, it cannot be undone or canceled. Undoing writes is a necessary property in designing distributed databases as one database server may agree to commit a transaction T but another may abort it, causing the first database server to cancel any of T's updates; failing to cancel such updates will break data consistency by allowing new transactions to read transaction updates that never committed. These challenging open problems makes this research project interesting.

Bibliography

[1] Divyakant Agrawal and Amr El Abbadi. "An efficient and fault-tolerant solution for distributed mutual exclusion". In: *ACM Transactions on Computer Systems (TOCS)* 9.1 (1991), pp. 1–20.

[2] Lindsey Allen, Panagiotis Antonopoulos, Arvind Arasu, Johannes Gehrke, Joachim Hammer, James Hunter, Raghav Kaushik, Donald Kossmann, Jonathan Lee, Ravi Ramamurthy, et al. "Veritas: Shared verifiable databases and tables in the cloud". In: CIDR. 2019.

[3] Gustavo Alonso and Amr El Abbadi. "Partitioned data objects in distributed databases". In: *Distributed and Parallel Databases* 3.1 (1995), pp. 5–35.

[4] *Amazon AWS*. https://aws.amazon.com/. Accessed: 2022-4-25.

[5] *Amazon AWS Account Hierarchy*. https://aws.amazon.com/answers/account-management/aws-multi-account-billing-strategy/. Accessed: 2020-02-17.

[6] *Amazon loses 1% revenue for every 100ms page load delay*. https://www.contentkingapp.com/academy/page-speed-resources/faq/amazon-page-speed-study/. Accessed May 9, 2022.

[7] *Amazon S3 Bucket Breaches*. https://www.riskiq.com/blog/labs/magecart-amazon-s3-buckets/. Accessed: 2019-07-10.

[8] MJ. Amiri, Sujaya Maiyya, D. Agrawal, and A. El Abbadi. "SeeMoRe: A Fault-Tolerant Protocol for Hybrid Cloud Environments". In: *International Conference on Data Engineering (ICDE)* (2020).

[9] MJ. Amiri, D. Shu, Sujaya Maiyya, D. Agrawal, and A. El Abbadi. *Ziziphus: Scalable Data Management Across Byzantine Edge Servers*. https://sites.cs.ucsb.edu/~sujaya_maiyya/assets/papers/ziziphus.pdf. Accessed: 2021-11-11. 2021.

[10] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. "ParBlockchain: Leveraging Transaction Parallelism in Permissioned Blockchain Systems". In: *39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2019.

[11] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. "Hyperledger fabric: a distributed operating system for permissioned blockchains". In: *Proceedings of the Thirteenth EuroSys Conference*. ACM. 2018, p. 30.

[12] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. *Above the clouds: A berkeley view of cloud computing*. Tech. rep. Technical Report UCB/EECS-2009-28, EECS Department, University of California ..., 2009.

[13] V. Arora, RKS. Babu, Sujaya Maiyya, D. Agrawal, A. El Abbadi, X. Xue, et al. "Dynamic Timestamp Allocation for Reducing Transaction Aborts". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 269–276.

[14] *Azure CosmosDB*. https://azure.microsoft.com/en-us/blog/a-technical-overview-of-azure-cosmos-db/. Accessed: 2020-06-17.

[15] *Azure Public Dataset*. https://github.com/Azure/AzurePublicDataset. 2019 (accessed June 30, 2020).

[16] Daniel Barbara and Hector Garcia-Molina. "The Demarcation Protocol: A technique for maintaining linear arithmetic constraints in distributed database systems". In: *International Conference on Extending Database Technology*. Springer. 1992, pp. 373–388.

[17] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. "Web search for a planet: The Google cluster architecture". In: *IEEE micro* 23.2 (2003), pp. 22–28.

[18] Luiz André Barroso and Urs Hölzle. "The datacenter as a computer: An introduction to the design of warehouse-scale machines". In: *Synthesis lectures on computer architecture* 4.1 (2009), pp. 1–108.

[19] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. "Chainspace: A sharded smart contracts platform". In: *arXiv preprint arXiv:1708.03778* (2017).

[20] Donald Beaver, Silvio Micali, and Phillip Rogaway. "The round complexity of secure protocols". In: *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. 1990, pp. 503–513.

[21] Josh Benaloh. "Dense probabilistic encryption". In: *Proceedings of the workshop on selected areas of cryptography*. 1994, pp. 120–128.

[22] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley Reading, 1987.

[23] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. "Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015, pp. 837–849.

[24] Robert Birke, Ioana Giurgiu, Lydia Y Chen, Dorothea Wiesmann, and Ton Engbersen. "Failure analysis of virtual and physical machines: patterns, causes and characteristics". In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE. 2014, pp. 1–12.

[25] Dan Boneh, Eyal Kushilevitz, Rafail Ostrovsky, and William E Skeith. "Public key encryption that allows PIR queries". In: *Annual International Cryptology Conference*. Springer. 2007, pp. 50–67.

[26] Dan Boneh, David Mazieres, and Raluca Ada Popa. "Remote oblivious storage: Making oblivious RAM practical". In: (2011).

[27] Elette Boyle, Kai-Min Chung, and Rafael Pass. "Oblivious parallel RAM and applications". In: *Theory of Cryptography Conference*. Springer. 2016, pp. 175–204.

[28] Elette Boyle and Moni Naor. "Is there an oblivious ram lower bound?" In: *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*. 2016, pp. 357–368.

[29] Gabriel Bracha and Sam Toueg. "Asynchronous consensus and broadcast protocols". In: *Journal of the ACM (JACM)* 32.4 (1985), pp. 824–840.

[30] Zvika Brakerski. "Fully homomorphic encryption without modulus switching from classical GapSVP". In: *Annual Cryptology Conference*. Springer. 2012, pp. 868–886.

[31] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. "TAO: Facebook's Distributed Data Store for the Social Graph". In: *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 2013, pp. 49–60.

[32] Christian Cachin, Silvio Micali, and Markus Stadler. "Computationally private information retrieval with polylogarithmic communication". In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 1999, pp. 402–414.

[33] David Cash, Andrew Drucker, and Alexander Hoover. "A lower bound for one-round oblivious RAM". In: *Theory of Cryptography Conference*. Springer. 2020, pp. 457–485.

[34] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. "Leakage-abuse attacks against searchable encryption". In: *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*. 2015, pp. 668–679.

[35] Miguel Castro, Barbara Liskov, et al. "Practical Byzantine fault tolerance". In: *OSDI*. Vol. 99. 1999. 1999, pp. 173–186.

[36] Anrin Chakraborti and Radu Sion. "ConcurORAM: High-throughput stateless parallel multi-client ORAM". In: *arXiv preprint arXiv:1811.04366* (2018).

[37] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective". In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. 2007, pp. 398–407.

[38] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. "Bigtable: A distributed storage system for structured data". In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), pp. 1–26.

[39] Bernadette Charron-Bost. "Comparing the atomic commitment and consensus problems". In: *Future directions in distributed computing*. Springer, 2003, pp. 29–34.

[40] Batyr Charyyev, Engin Arslan, and Mehmet Hadi Gunes. "Latency comparison of cloud datacenters and edge servers". In: *GLOBECOM 2020-2020 IEEE Global Communications Conference*. IEEE. 2020, pp. 1–6.

[41] Zhijia Chen, Yuanchang Zhu, Yanqiang Di, and Shaochong Feng. "Self-adaptive prediction of cloud resource demands using ensemble model and subtractive-fuzzy clustering based fuzzy neural network". In: *Computational intelligence and neuroscience* 2015 (2015).

[42] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. "Private information retrieval". In: *Proceedings of IEEE 36th Annual Foundations of Computer Science*. IEEE. 1995, pp. 41–50.

[43] Richard Chow, Philippe Golle, Markus Jakobsson, Elaine Shi, Jessica Staddon, Ryusuke Masuoka, and Jesus Molina. "Controlling data in the cloud: outsourcing computation without outsourcing control". In: *Proceedings of the 2009 ACM workshop on Cloud computing security*. 2009, pp. 85–90.

[44] 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*. ACM. 2010, pp. 143–154.

[45] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. "Spanner: Google's globally distributed database". In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), pp. 1–22.

[46] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 153–167.

[47] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. "Obladi: Oblivious serializable transactions in the cloud". In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 727–743.

[48] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. "Schism: a workload-driven approach to database replication and partitioning". In: (2010).

[49] *Cyber Threat Data Manipulation*. https://www.theguardian.com/technology/2015/sep/10/cyber-threat-data-manipulation-us-intelligence-chief. Accessed: 2019-07-10.

[50] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. "Snoopy: Surpassing the Scalability Bottleneck of Oblivious Storage". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 655–671.

[51] Jonathan Dautrich, Emil Stefanov, and Elaine Shi. "Burst {ORAM}: Minimizing {ORAM} Response Times for Bursty Access Patterns". In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 749–764.

[52] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: amazon's highly available key-value store". In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 205–220.

[53] *Default Replica Count For CockroahDB*. https://www.cockroachlabs.com/docs/stable/configure-replication-zones.html. Accessed Jan 10, 2021.

[54] *Default Replica Count For Spanner*. https://cloud.google.com/spanner/docs/instances. Accessed Jan 10, 2021.

[55] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. "{SEAL}: Attack Mitigation for Encrypted Databases via Adjustable Leakage". In: *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2020, pp. 2433–2450.

[56] Sheng Di, Derrick Kondo, and Walfredo Cirne. "Host load prediction in a Google compute cloud with a Bayesian model". In: *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2012, pp. 1–11.

[57] Danny Dolev and H. Raymond Strong. "Authenticated algorithms for Byzantine agreement". In: *SIAM Journal on Computing* 12.4 (1983), pp. 656–666.

[58] Taher ElGamal. "A public key cryptosystem and a signature scheme based on discrete logarithms". In: *IEEE transactions on information theory* 31.4 (1985), pp. 469–472.

[59] *Equifax Data Breach*. https://investor.equifax.com/news-and-events/news/2017/09-15-2017-224018832. Accessed: 2017-09-15.

[60] *Facebook fined $5B over data privacy violation*. https://www.ftc.gov/news-events/press-releases/2019/07/ftc-imposes-5-billion-penalty-sweeping-new-privacy-restrictions. Accessed June 7, 2021.

[61] Junfeng Fan and Frederik Vercauteren. "Somewhat practical fully homomorphic encryption." In: *IACR Cryptol. ePrint Arch.* 2012 (2012), p. 144.

[62] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. "Impossibility of distributed consensus with one faulty process". In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382.

[63] Christopher Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. "Bucket ORAM: single online roundtrip, constant bandwidth oblivious RAM". In: *Cryptology ePrint Archive* (2015).

[64] Daniel Ford, François Labelle, Florentina I Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. "Availability in globally distributed storage systems". In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. 2010.

[65] *FTC imposes 5 billion dollars fine for privacy violations*. https://www.ftc.gov/news-events/news/press-releases/2019/07/ftc-imposes-5-billion-penalty-sweeping-new-privacy-restrictions-facebook. Accessed: 2022-4-25.

[66] Edoardo Gaetani, Leonardo Aniello, Roberto Baldoni, Federico Lombardi, Andrea Margheri, and Vladimiro Sassone. "Blockchain-based database to ensure data integrity in cloud computing environments". In: *ITA-SEC* (2017).

[67] Rui Garcia, Rodrigo Rodrigues, and Nuno Preguiça. "Efficient middleware for byzantine fault tolerant database replication". In: *European Conference on Computer Systems (EuroSys)*. ACM. 2011, pp. 107–122.

[68] Hector Garcia Molina, Frank Pittelli, and Susan Davidson. "Applications of Byzantine agreement in database systems". In: *ACM Transactions on Database Systems (TODS)* 11.1 (1986), pp. 27–47.

[69] Hector Garcia-Molina and Kenneth Salem. "Main memory database systems: An overview". In: *IEEE Transactions on knowledge and data engineering* 4.6 (1992), pp. 509–516.

[70] Sanjam Garg, Steve Lu, Rafail Ostrovsky, and Alessandra Scafuro. "Garbled RAM from one-way functions". In: *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*. 2015, pp. 449–458.

[71] Ilir Gashi, Peter Popov, Vladimir Stankovic, and Lorenzo Strigini. "On designing dependable services with diverse off-the-shelf SQL servers". In: *Architecting Dependable Systems II*. Springer, 2004, pp. 191–214.

[72] *GDPR Regulations*. https://gdpr-info.eu/. Accessed May 10, 2022.

[73] Craig Gentry et al. *A fully homomorphic encryption scheme*. Vol. 20. 9. Stanford university Stanford, 2009.

[74] Craig Gentry, Shai Halevi, Steve Lu, Rafail Ostrovsky, Mariana Raykova, and Daniel Wichs. "Garbled RAM revisited". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2014, pp. 405–422.

[75] Craig Gentry and Zulfikar Ramzan. "Single-database private information retrieval with constant communication rate". In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2005, pp. 803–815.

[76] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. "Scalable consistency in Scatter". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 15–28.

[77] Oded Goldreich. "Towards a theory of software protection and simulation by oblivious RAMs". In: *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 1987, pp. 182–194.

[78] Oded Goldreich and Rafail Ostrovsky. "Software protection and simulation on oblivious RAMs". In: *Journal of the ACM (JACM)* 43.3 (1996), pp. 431–473.

[79] Leana Golubchik and Alexander Thomasian. "Token allocation in distributed systems". In: *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*. IEEE. 1992, pp. 64–71.

[80] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. "Press: Predictive elastic resource scaling for cloud systems". In: *2010 International Conference on Network and Service Management*. Ieee. 2010, pp. 9–16.

[81] *Google Cloud*. https://cloud.google.com/. Accessed: 2022-4-25.

[82] *Google Cloud Enterprise Hierarchy*. https://cloud.google.com/docs/enterprise/best-practices-for-enterprise-organizations. Accessed: 2020-02-17.

[83] *Google Cloud Pricing*. https://cloud.google.com/storage/pricing. Accessed August 15, 2021.

[84] *Google fined $57M over GDPR violation*. https://digitalguardian.com/blog/google-fined-57m-data-protection-watchdog-over-gdpr-violations. Accessed June 7, 2021.

[85] *Google Function Pricing*. https://cloud.google.com/functions/pricing. Accessed August 15, 2021.

[86] *Google loses 20% traffic for 0.5s page load delay*. https://medium.com/@vikigreen/impact-of-slow-page-load-time-on-website-performance-40d5c9ce568a. Accessed May 9, 2022.

[87] James N Gray. "Notes on data base operating systems". In: *Operating Systems*. Springer, 1978, pp. 393–481.

[88] Jim Gray and Leslie Lamport. "Consensus on transaction commit". In: *ACM Transactions on Database Systems (TODS)* 31.1 (2006), pp. 133–160.

[89] *Growth of Global Data Privacy Laws*. https://stealthbits.com/blog/growth-of-global-data-privacy-laws/. Accessed: 2021-10-25.

[90] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. "Pancake: Frequency smoothing for encrypted data stores". In: *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2020, pp. 2451–2468.

[91] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. "Learning to reconstruct: Statistical learning theory and encrypted database attacks". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 1067–1083.

[92] Rachid Guerraoui. "Revisiting the relationship between non-blocking atomic commitment and consensus". In: *International Workshop on Distributed Algorithms*. Springer. 1995, pp. 87–100.

[93] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. "Why does the cloud stop computing? lessons from hundreds of service outages". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 2016, pp. 1–16.

[94] Harshit Gupta and Umakishore Ramachandran. "Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access". In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. 2018, pp. 148–159.

[95] Vassos Hadzilacos. "On The Relationship Between The Atomic Commitment And Consensus Problems". In: *In Fault-Tolerant Distributed Computing*. Springer-Verlag, 1990, pp. 201–208.

[96] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. "PeerReview: Practical accountability for distributed systems". In: *ACM SIGOPS operating systems review* 41.6 (2007), pp. 175–188.

[97] Theo Härder. "Handling hot spot data in DB-sharing systems". In: *Information Systems* 13.2 (1988), pp. 155–166.

[98] Maurice P Herlihy and Jeannette M Wing. "Linearizability: A correctness condition for concurrent objects". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492.

[99] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. "BlockchainDB: a shared database on blockchains". In: *Proceedings of the VLDB Endowment* 12.11 (2019), pp. 1597–1609.

[100] *HIPAA Regulations*. https://www.hhs.gov/hipaa/index.html. Accessed May 10, 2022.

[101] *How many servers does a data center have?* https://www.racksolutions.com/news/blog/how-many-servers-does-a-data-center-have/. Accessed August 15, 2021.

[102] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. "Flexible paxos: Quorum intersection revisited". In: *arXiv preprint arXiv:1608.06696* (2016).

[103] *Impact of slow page load time*. https://medium.com/@vikigreen/impact-of-slow-page-load-time-on-website-performance-40d5c9ce568a. Accessed: 2021-06-6.

[104] *Increasing Cyber-attacks on Cloud Services*. https://compliancy-group.com/cyber-attacks-on-cloud-services-rise-630/. Accessed: 2021-10-25.

[105] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. "Access pattern disclosure on searchable encryption: ramification, attack and mitigation." In: *Ndss*. Vol. 20. Citeseer. 2012, p. 12.

[106] Rohit Jain and Sunil Prabhakar. "Trustworthy data from untrusted databases". In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE. 2013, pp. 529–540.

[107] Yexi Jiang, Chang-shing Perng, Tao Li, and Rong Chang. "Asap: A self-adaptive prediction system for instant cloud resource demand provisioning". In: *2011 IEEE 11th International Conference on Data Mining*. IEEE. 2011, pp. 1104–1109.

[108] Antonios Katsarakis, Vasilis Gavrielatos, MR Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. "Hermes: a fast, fault-tolerant and linearizable replication protocol". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 201–217.

[109] Idit Keidar and Danny Dolev. "Increasing the resilience of atomic commit, at no additional cost". In: *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 1995, pp. 245–254.

[110] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. "Generic attacks on secure outsourced databases". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 2016, pp. 1329–1340.

[111] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. "Enhancing bitcoin security and performance with strong consistency via collective signing". In: *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 2016, pp. 279–296.

[112] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. "Omniledger: A secure, scale-out, decentralized ledger via sharding". In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 583–598.

[113] Richard Koo and Sam Toueg. "Checkpointing and rollback-recovery for distributed systems". In: *IEEE Transactions on software Engineering* 1 (1987), pp. 23–31.

[114] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. "Data recovery on encrypted databases with k-nearest neighbor query leakage". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 1033–1050.

[115] Kari Korpela, Jukka Hallikas, and Tomi Dahlberg. "Digital supply chain transformation toward blockchain integration". In: *proceedings of the 50th Hawaii international conference on system sciences*. 2017.

[116] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. "MDCC: Multi-data center consistency". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. 2013, pp. 113–126.

[117] Narayanan Krishnakumar and Arthur J Bernstein. "High throughput escrow algorithms for replicated databases". In: *VLDB*. Vol. 1992. 1992, pp. 175–186.

[118] Akhil Kumar and Michael Stonebraker. "Semantics based transaction management techniques for replicated data". In: *ACM SIGMOD Record* 17.3 (1988), pp. 117–125.

[119] Hsiang-Tsung Kung and John T Robinson. "On optimistic methods for concurrency control". In: *ACM Transactions on Database Systems (TODS)* 6.2 (1981), pp. 213–226.

[120] Eyal Kushilevitz and Rafail Ostrovsky. "Replication is not needed: Single database, computationally-private information retrieval". In: *Proceedings 38th annual symposium on foundations of computer science*. IEEE. 1997, pp. 364–373.

[121] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* 44.2 (2010), pp. 35–40.

[122] Leslie Lamport et al. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[123] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), pp. 558–565.

[124] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. "Stoppable paxos". In: *TechReport, Microsoft Research* (2008).

[125] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. "Vertical paxos and primary-backup replication". In: *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM. 2009, pp. 312–313.

[126] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine generals problem". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401.

[127] Butler Lampson and David Lomet. "A new presumed commit optimization for two phase commit". In: *19th International Conference on Very Large Data Bases (VLDB'93)*. 1993, pp. 630–640.

[128] Butler Lampson and Howard E Sturgis. "Crash recovery in a distributed data storage system". In: (1979).

[129] Kasper Green Larsen, Tal Malkin, Omri Weinstein, and Kevin Yeo. "Lower bounds for oblivious near-neighbor search". In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2020, pp. 1116–1134.

[130] Kasper Green Larsen and Jesper Buus Nielsen. "Yes, there is an oblivious RAM lower bound!" In: *Annual International Cryptology Conference*. Springer. 2018, pp. 523–542.

[131] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. "Dynamic authenticated index structures for outsourced databases". In: *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM. 2006, pp. 121–132.

[132] Yehuda Lindell and Benny Pinkas. "A proof of security of Yao's protocol for two-party computation". In: *Journal of cryptology* 22.2 (2009), pp. 161–188.

[133] Bruce G Lindsay, Patricia G Selinger, Cesare Galtieri, James N Gray, Raymond A Lorie, Thomas G Price, Franco Putzolu, Irving L Traiger, and Bradford W Wade. *Notes on distributed databases*. IBM Thomas J. Watson Research Division, 1979.

[134] Helger Lipmaa and Bingsheng Zhang. "Two new efficient PIR-writing protocols". In: *International Conference on Applied Cryptography and Network Security*. Springer. 2010, pp. 438–455.

[135] Gang Liu, Kenli Li, Zheng Xiao, and Rujia Wang. "EHAP-ORAM: Efficient Hardware-Assisted Persistent ORAM System for Non-volatile Memory". In: *arXiv preprint arXiv:2011.03669* (2020).

[136] Zheli Liu, Bo Li, Yanyu Huang, Jin Li, Yang Xiang, and Witold Pedrycz. "NewMCOS: towards a practical multi-cloud oblivious storage scheme". In: *IEEE Transactions on Knowledge and Data Engineering* 32.4 (2019), pp. 714–727.

[137] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. "Stronger Semantics for Low-Latency Geo-Replicated Storage". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 2013, pp. 313–328.

[138] Steve Lu and Rafail Ostrovsky. "Distributed oblivious RAM for secure two-party computation". In: *Theory of Cryptography Conference*. Springer. 2013, pp. 377–396.

[139] Steve Lu and Rafail Ostrovsky. "How to garble RAM programs?" In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2013, pp. 719–734.

[140] Aldelir Fernando Luiz, Lau Cheuk Lung, and Miguel Correia. "Byzantine fault-tolerant transaction processing for replicated databases". In: *2011 IEEE 10th International Symposium on Network Computing and Applications*. IEEE. 2011, pp. 83–90.

[141] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. "A secure sharding protocol for open blockchains". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 17–30.

[142] N. A. Lynch and A. A. Shvartsman. "Robust Emulation of Shared Memory Using Dynamic Quorum-acknowledged Broadcasts". In: *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*. FTCS '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 272–. ISBN: 0-8186-7831-3. URL: http://dl.acm.org/citation.cfm?id=795670.796859.

[143] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. "Low-latency multi-datacenter databases using replicated commit". In: *Proceedings of the VLDB Endowment* 6.9 (2013), pp. 661–672.

[144] Sujaya Maiyya, I. Ahmad, D. Agrawal, and A. El Abbadi. "Samya: Geo-Distributed Data System for High Contention Data Aggregates". In: *International Conference on Data Engineering (ICDE)* (2021).

[145] Sujaya Maiyya, P. Ananth, D. Agrawal, and A. El Abbadi. *ORTOA: One Round Trip Oblivious Access*. https://sites.cs.ucsb.edu/~sujaya_maiyya/assets/papers/ORTOA.pdf. Accessed: 2021-11-11.

[146] Sujaya Maiyya, DBH. Cho, D. Agrawal, and A. El Abbadi. "Fides: Managing Data on Untrusted Infrastructure". In: *International Conference on Distributed Computing Systems (ICDCS)* (2020).

[147] Sujaya Maiyya, Seif Ibrahim, Caitlin Scarberry, Divyakant Agrawal, Amr El Abbadi, Huijia Lin, Stefano Tessaro, and Victor Zakhary. "QuORAM: A Quorum-Replicated Fault Tolerant ORAM Datastore". In: *To appear in USENIX Security* (2022).

[148] Sujaya Maiyya, F. Nawab, D. Agrawal, and A. El Abbadi. "Unifying consensus and atomic commitment for effective cloud data management". In: *Proceedings of the VLDB Endowment* 12.5 (2019), pp. 611–623.

[149] Sujaya Maiyya, V. Zakhary, D. Agrawal, and A. El Abbadi. “Database and distributed computing fundamentals for scalable, fault-tolerant, and consistent maintenance of blockchains”. In: *Proceedings of the VLDB Endowment* 11.12 (2018), pp. 2098–2101.

[150] Sujaya Maiyya, V. Zakhary, MJ. Amiri, D. Agrawal, and A. El Abbadi. “Database and Distributed Computing Foundations of Blockchains”. In: *SIGMOD*. 2019.

[151] Trent McConaghy, Rodolphe Marques, Andreas Müller, Dimitri De Jonghe, Troy McConaghy, Greg McMullen, Ryan Henderson, Sylvain Bellemare, and Alberto Granzotto. “BigchainDB: a scalable blockchain database”. In: *white paper, BigChainDB* (2016).

[152] Ralph C Merkle. “A certified digital signature”. In: *Conference on the Theory and Application of Cryptology*. Springer. 1989, pp. 218–238.

[153] *Microsoft Azure*. https://azure.microsoft.com/en-us/. Accessed: 2022-4-25.

[154] *Microsoft Azure Resource Hierarchy*. https://docs.microsoft.com/en-us/azure/cloud-adoption-framework/ready/azure-setup-guide/organize-resources?tabs=AzureManagmentGroupsAndHierarchy. Accessed: 2020-02-17.

[155] *Microsoft SEAL*. https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/homomorphic-encryption-seal. Accessed June 15, 2021.

[156] *Microsoft’s attention span study*. https://dl.motamem.org/microsoft-attention-spans-research-report.pdf. Accessed: 2022-4-25.

[157] 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”. In: *ACM Transactions on Database Systems (TODS)* 17.1 (1992), pp. 94–162.

[158] C Mohan, Ray Strong, and Shel Finkelstein. “Method for distributed transaction commit and recovery using Byzantine agreement within clusters of processors”. In: *Proceedings of the second annual ACM symposium on Principles of distributed computing*. 1983, pp. 89–103.

[159] *MongoDB*. https://www.mongodb.com/. Accessed March 14, 2022.

[160] Iulian Moraru, David G Andersen, and Michael Kaminsky. “There is more consensus in egalitarian parliaments”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 358–372.

[161] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. “Consolidating concurrency control and consensus for commits under conflicts”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 517–532.

[162] Satoshi Nakamoto et al. “Bitcoin: A peer-to-peer electronic cash system”. In: (2008).

[163] Moni Naor and Udi Wieder. “Scalable and dynamic quorum systems”. In: *Distributed Computing* 17.4 (2005), pp. 311–322.

[164] Maithili Narasimha and Gene Tsudik. “Authentication of outsourced databases using signature aggregation and chaining”. In: *International conference on database systems for advanced applications*. Springer. 2006, pp. 420–436.

[165] Arvind Narayanan and Vitaly Shmatikov. “Myths and fallacies of” personally identifiable information””. In: *Communications of the ACM* 53.6 (2010), pp. 24–26.

[166] Arvind Narayanan and Vitaly Shmatikov. “Robust de-anonymization of large sparse datasets”. In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE. 2008, pp. 111–125.

[167] *Neflix uses AWS for all compute and storage needs*. https://aws.amazon.com/solutions/case-studies/netflix/. Accessed October 5, 2021.

[168] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. “{AGILE}: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service”. In: *Proceedings of the 10th International Conference on Autonomic Computing ({ICAC} 13)*. 2013, pp. 69–82.

[169] Jakob Nielsen. *Usability engineering*. Morgan Kaufmann, 1994.

[170] Rodrigo Nogueira, Filipe Araújo, and Raul Barbosa. “CloudBFT: elastic byzantine fault tolerance”. In: *2014 IEEE 20th Pacific Rim International Symposium on Dependable Computing*. IEEE. 2014, pp. 180–189.

[171] Patrick E O’Neil. “The escrow transactional method”. In: *ACM Transactions on Database Systems (TODS)* 11.4 (1986), pp. 405–430.

[172] Diego Ongaro and John Ousterhout. “In search of an understandable consensus algorithm”. In: *2014 USENIX Annual Technical Conference (Usenix ATC 14)*. 2014, pp. 305–319.

[173] Pascal Paillier. “Public-key cryptosystems based on composite degree residuosity classes”. In: *International conference on the theory and applications of cryptographic techniques*. Springer. 1999, pp. 223–238.

[174] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. “Pinocchio: Nearly practical verifiable computation”. In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 238–252.

[175] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. “What Storage Access Privacy is Achievable with Small Overhead?” In: *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 2019, pp. 182–199.

[176] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 2012, pp. 61–72.

[177] Fernando Pedone and Nicolas Schiper. "Byzantine fault-tolerant deferred update replication". In: *Journal of the Brazilian Computer Society* 18.1 (2012), p. 3.

[178] Giuseppe Persiano and Kevin Yeo. "Lower bounds for differentially private RAMs". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2019, pp. 404–434.

[179] Raluca Ada Popa, Catherine MS Redfield, Nickolai Zeldovich, and Hari Balakrishnan. "CryptDB: Protecting confidentiality with encrypted query processing". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 85–100.

[180] *Redis*. https://redis.io/. Accessed March 14, 2022.

[181] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. "Constants Count: Practical Improvements to Oblivious {RAM}". In: *24th USENIX Security Symposium ({USENIX} Security 15)*. 2015, pp. 415–430.

[182] *Resource Tracking Services.* https://thedigitalprojectmanager.com/resource-scheduling-software-tools/. Accessed Oct 3, 2020).

[183] Ronald L Rivest, Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[184] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. "Taostore: Overcoming asynchronicity in oblivious data storage". In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 198–217.

[185] Claus-Peter Schnorr. "Efficient signature generation by smart cards". In: *Journal of cryptology* 4.3 (1991), pp. 161–174.

[186] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. "Clay: fine-grained adaptive partitioning for general database schemas". In: *Proceedings of the VLDB Endowment* 10.4 (2016), pp. 445–456.

[187] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. "Conflict-free replicated data types". In: *Symposium on Self-Stabilizing Systems*. Springer. 2011, pp. 386–400.

[188] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, et al. "F1: the fault-tolerant distributed RDBMS supporting google's ad business". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 2012, pp. 777–778.

[189] Dale Skeen. "Nonblocking commit protocols". In: *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. ACM. 1981, pp. 133–142.

[190] Dale Skeen and Michael Stonebraker. "A formal model of crash recovery in a distributed system". In: *IEEE Transactions on Software Engineering* 3 (1983), pp. 219–228.

[191] *Spotify backend infrastructure moves to Google Cloud*. https://variety.com/2016/digital/news/spotify-goes-cloud-no-more-data-centers-1201712891/. Accessed October 5, 2021.

[192] Emil Stefanov and Elaine Shi. "Multi-cloud oblivious storage". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, pp. 247–258.

[193] Emil Stefanov and Elaine Shi. "Oblivistore: High performance oblivious cloud storage". In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 253–267.

[194] Emil Stefanov, Elaine Shi, and Dawn Song. "Towards practical oblivious RAM". In: *arXiv preprint arXiv:1106.3652* (2011).

[195] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. "Path ORAM: an extremely simple oblivious RAM protocol". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, pp. 299–310.

[196] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. "Keeping authorities" honest or bust" with decentralized witness cosigning". In: *2016 IEEE Symposium on Security and Privacy (SP)*. Ieee. 2016, pp. 526–545.

[197] 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". In: *Proceedings of the VLDB Endowment* 8.3 (2014), pp. 245–256.

[198] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. "CockroachDB: The Resilient Geo-Distributed SQL Database". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 1493–1509.

[199] *Tech companies face billions in fines under new privacy laws.* https://news.bloomberglaw.com/privacy-and-data-security/tech-companies-face-billions-in-fines-under-eu-content-rules. Accessed: 2022-4-25.

[200] Jeff Terrace and Michael J Freedman. "Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads." In: *USENIX Annual Technical Conference*. June. San Diego, CA. 2009, pp. 1–16.

[201] Robert H Thomas. "A majority consensus approach to concurrency control for multiple copy databases". In: *ACM Transactions on Database Systems (TODS)* 4.2 (1979), pp. 180–209.

[202] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. "Calvin: fast distributed transactions for partitioned database systems". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM. 2012, pp. 1–12.

[203] *Top reasons to migrate to cloud.* https://www.forbes.com/sites/forbestechcouncil/2021/03/12/why-migrate-to-the-cloud-the-basics-benefits-and-real-life-examples/?sh=ec5f3c65e272. Accessed: 2022-4-25.

[204] Shruti Tople, Yaoqi Jia, and Prateek Saxena. "Pro-oram: Practical read-only oblivious {RAM}". In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*. 2019, pp. 197–211.

[205] *TSL.* https://datatracker.ietf.org/doc/html/rfc5246. Accessed April 14, 2022.

[206] *Twitter selects AWS to power user feeds.* https://press.aboutamazon.com/news-releases/news-release-details/twitter-selects-aws-strategic-provider-serve-timelines/. Accessed October 5, 2021.

[207] *User's perception of performance delays.* https://web.dev/rail/. Accessed: 2022-4-25.

[208] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. "Tolerating byzantine faults in transaction processing systems using commit barrier scheduling". In: *ACM SIGOPS Operating Systems Review*. Vol. 41. ACM. 2007, pp. 59–72.

[209] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. "Amazon aurora: Design considerations for high throughput cloud-native relational databases". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 1041–1052.

[210] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. "Characterizing cloud computing hardware reliability". In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 193–204.

[211] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. “Splinter: Practical private queries on public data”. In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, pp. 299–313.

[212] Guosai Wang, Lifei Zhang, and Wei Xu. “What can we learn from four years of data center hardware failures?” In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2017, pp. 25–36.

[213] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[214] Mor Weiss and Daniel Wichs. “Is there an oblivious RAM lower bound for online reads?” In: *Journal of Cryptology* 34.3 (2021), pp. 1–44.

[215] Peter Williams and Radu Sion. “Single round access privacy on outsourced storage”. In: *Proceedings of the 2012 ACM conference on Computer and communications security*. 2012, pp. 293–304.

[216] Peter Williams, Radu Sion, and Alin Tomescu. “Privatefs: A parallel oblivious file system”. In: *Proceedings of the 2012 ACM conference on Computer and communications security*. 2012, pp. 977–988.

[217] Andrew Chi-Chih Yao. “How to generate and exchange secrets”. In: *27th Annual Symposium on Foundations of Computer Science*. IEEE. 1986, pp. 162–167.

[218] Sergey Yekhanin. “Private information retrieval”. In: *Locally Decodable Codes and Private Information Retrieval Schemes*. Springer, 2010, pp. 61–74.

[219] Aydan R Yumerefendi and Jeffrey S Chase. “Strong accountability for network storage”. In: *ACM Transactions on Storage (TOS)* 3.3 (2007), p. 11.

[220] Aydan R Yumerefendi and Jeffrey S Chase. “The role of accountability in dependable distributed systems”. In: *Proceedings of HotDep*. Vol. 5. Citeseer. 2005, pp. 3–3.

[221] Aydan R Yumerefendi and Jeffrey S Chase. “Trust but verify: accountability for network services”. In: *Proceedings of the 11th workshop on ACM SIGOPS European workshop*. ACM. 2004, p. 37.

[222] V. Zakhary, MJ Amiri, Sujaya Maiyya, D. Agrawal, and A. El Abbadi. “Towards Global Asset Management in Blockchain Systems”. In: *BCDL co-located with VLDB* (2019).

[223] Victor Zakhary, Cetin Sahin, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. “Pharos: Privacy Hazards of Replicating ORAM Stores”. In: *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018*. 2018.

[224] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. "Rapidchain: Scaling blockchain via full sharding". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 931–948.

[225] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. "Chiller: Contention-centric Transaction Execution and Data Partitioning for Modern Networks". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 511–526.

[226] Honglei Zhang, Hua Chai, Wenbing Zhao, P Michael Melliar-Smith, and Louise E Moser. "Trustworthy coordination of Web services atomic transactions". In: *IEEE Transactions on Parallel and Distributed Systems* 23.8 (2011), pp. 1551–1565.

[227] Irene Zhang, Naveen Kr Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan RK Ports. "Building consistent transactions with inconsistent replication". In: *ACM Transactions on Computer Systems (TOCS)* 35.4 (2018), pp. 1–37.

[228] Jinsheng Zhang, Qiumao Ma, Wensheng Zhang, and Daji Qiao. "TSKT-ORAM: A two-server k-ary tree ORAM for access pattern protection in cloud storage". In: *MILCOM 2016-2016 IEEE Military Communications Conference*. IEEE. 2016, pp. 527–532.

[229] F. Zhao, Sujaya Maiyya, R. Wiener, D. Agrawal, and A. El Abbadi. "KLL$^{\pm}$: Approximate Quantile Sketches over Dynamic Datasets". In: *Proceedings of the VLDB Endowment* (2021).

[230] Wenbing Zhao. "A byzantine fault tolerant distributed commit protocol". In: *Third IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC 2007)*. IEEE. 2007, pp. 37–46.