Restart-Rollback: A Fault model for distributed systems with persistent state

Baltasar Azevedo e Silva Dinis

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Rodrigo Seromenho Miragaia Rodrigues
Prof. Peter Druschel

Examination Committee

Chairperson: Prof. José Alberto Rodrigues Pereira Sardinha
Supervisor: Prof. Rodrigo Seromenho Miragaia Rodrigues
Member of the Committee: Prof. Nuno Miguel Carvalho dos Santos

November 2022

Acknowledgments

I would like to thank Professor Rodrigo Rodrigues, not only for his supervision of this thesis but for over four years of mentoring and countless doors he has opened for me. Without a doubt, my academic, professional and personal development would not have been the same without his intervention. I would also like to thank Professor Peter Druschel, who welcomed me multiple times at the Max Planck Institute for Software Systems in Saarbrücken, an institution where I learned a lot of lessons which I will carry with me and for his astute comments and scientific rigour, from which I have learned a lot.

I would like to acknowledge several other colleagues, mentors and contributors, who have been with me through my academic journey: Afonso Tinoco, Roberta De Viti, Professor Deepak Garg, Professor Vijay Chidambaram, Professor Bobby Bhattacharjee and Professor Pedro Adão.

I would like to thank my parents, for the housing and food they have provided me through the years, an indispensible prerequisite for good research. Also, for raising me.

Finally, I would like to thank my friends. Although words are not nearly enough to describe the shared moments (both of joy and hardship), the companionship, the comraderie and the bond that connects us, I am forever grateful: Afonso, Andreia, Chico, Cintrão, Donga, Filipa, Gaspar, Homem, João, Lamego, Lé, Maria, Mariana, Marta Ambrósio, Marta Sousa, Pacheco, Pedro, Pina, Regouga, Reynolds, Rocha, Sanches, Vasco, Zé and Leonor.

This work was supported by Fundação para a Ciência e a Tecnologia under project PTDC/CCI-INF/6762/2020.

Abstract

TEEs ensure the confidentiality and integrity of computations in hardware. Subject to the TEE's threat model, the hardware shields computation from most externally induced faults except crashes. As a result, CFT replication protocols should be sufficient when replicating TEEs. However, TEEs do not offer efficient means of ensuring the freshness of persistent state, which can be rolled back to an earlier version. In this dissertation, we propose the Restart-Rollback (RR) fault model for replicating TEEs, which precisely captures the possible fault behaviors of TEEs with external state, making it possible to avoid using expensive BFT protocols. Then, we show that existing CFT replication protocols can be easily adapted to this fault model with few changes, while retaining their original performance.

To illustrate the usefulness of TEE replication under RR, we built a replicated metadata service called TEEMS, which can be used to add TEE-grade confidentiality, integrity, and freshness to untrusted cloud storage. Then, we showcase the generality of the RR model, by applying it to the context of replicated KVSs. Current replicated KVSs make a tradeoff between latency, throughput and durability, depending on whether they flush writes before replying (which offers poor performance) or batch writes in memory (possibly losing data if nodes crash before flushing). We observe that the latter case effectively constitutes a rollback, which is captured by RR.

We implemented the various protocols and systems, and our evaluation shows performance that is comparable to CFT systems but with stronger guarantees, and significantly better than BFT systems.

Keywords

Cloud Storage, Distributed Storage, Distributed Systems, Durability, Fault Model, Persistence, Replication, Replication Protocol, Storage, Trusted Execution Environments

Resumo

Ambientes de Execução Confiável (em inglês, TEEs) garantem a confidencialidade e integridade de computações ao nível do *hardware*. Sobre o modelo adversarial dos TEEs, o *hardware* protege a computação da maioria das falhas induzidas externamente, excepto falhas de *crash*, em que um TEE simplesmente vai abaixo (e.g., quando falta a eletricidade). Contudo, TEEs, não possuem métodos eficientes de garantir a frescura do seu estado persistente, que pode ser revertido para uma versão anterior. Nesta dissertação, propomos o modelo de faltas Restart-Rollback (RR) para replicação de TEEs, que captura exatamente os comportamentos de falta de TEEs com estado externo, evitando a utilização de protocolos BFT, que incorrem num custo superior. Mostramos que protocolos CFT existentes podem ser adaptados facilmente para este modelo de faltas com poucas alterações, mantendo a sua performance original.

Para ilustrar a utilidade da replicação de TEEs segundo o modelo RR, construímos um serviço de metadados replicado chamado TEEMS, que pode ser usado para acrescentar confidencialidade, integridade e frescura a armazenamento em cloud ao nível das proteções dos TEEs. Seguidamente, exemplificamos a generalidade do modelo RR, aplicando-o ao contexto de KVSs replicadas. KVSs replicadas atuais escolhem entre latência, *throughput* e durabilidade, dependendo se sincronizam as escritas antes de responderem aos pedidos (o que oferece uma *performance* fraca) ou se combinam várias escritas numa *batch* em memória (possivelmente perdendo dados se os nós forem abaixo antes da escrita ser sincronizada). Observamos que o último caso efetivamente corresponde a uma reversão de estado, que é capturada pelo modelo RR.

Impelmentámos os vários protocolos e sistemas, e a nossa avaliação demonstra que a *performance* é comparável à de sistemas CFT mas com garantias mais fortes e significativamente superior à de sistemas BFT.

Palavras Chave

Ambientes de Execução Confiável, Armazenamento de Dados, Armazenamento de Dados Distribuído, Armazenamento em Nuvem, Durabilidade, Modelo de Faltas, Persistência, Protocolo de Replicação, Replicação, Sistemas Distribuídos

Contents

1 Introduction 1
1.1 Contributions 5
1.2 Organization of the Document 6
1.3 Publication 6
2 Related Work 7
2.1 Fault Models 9
2.2 Replication Protocols 11
2.3 Trusted Execution Environments 11
2.4 Single-node Key-Value Stores (KVSs) 14
2.4.1 Log-structured merge-trees (LSM-trees) 14
2.5 Persistence in replicated distributed systems 15
3 The Restart-Rollback (RR) Fault Model and Its Protocols 17
3.1 Motivation 19
3.1.1 Trusted Execution Environment (TEE) properties 19
3.1.2 Extending the usefulness of fault model 21
3.2 RR model definition 22
3.2.1 Replicated systems 22
3.2.2 Replication in the RR model 23
3.2.3 Deriving RR Quorums 23
3.3 Replication protocols 29
3.3.1 Principles and challenges for protocol adaptation 29
3.3.2 Read-write register 30
3.3.3 State-machine replication 34
3.3.4 Security Properties of TEE replication 37
3.4 Evaluation of TEE replication protocols 37
3.4.1 Implementation 37
3.4.2 Experimental Setup. 38

3.4.3 Code changes 38
3.4.4 Protocol performance in different models 38
3.5 Discussion: Generalization of the RR model 42

4 TEE replication using the RR model **43**
4.1 TEE Metadata System (TEEMS): Metadata service for trusted cloud storage 45
4.1.1 Motivation 45
4.1.2 TEE-grade cloud storage with TEEMS 46
4.1.3 Leveraging different storage protocols 47
4.1.4 Security Properties 48
4.1.5 Deployment scenarios 48
4.2 Implementation 49
4.3 Evaluation 50
4.3.1 TEEMS-based storage 50
4.3.2 TEEMS-based storage running Yahoo! Cloud Serving Benchmark (YCSB) 52

5 Leveraging the RR model in distributed storage **53**
5.1 Motivation 55
5.2 Restart-Rollback Storage System (R2-S2) Design 56
5.2.1 Asymmetric Synchronization 57
5.2.2 System Overview 57
5.2.3 Scheduling 58
5.2.4 Storage Layer 60
5.3 Parameterizing the System 61
5.4 Preliminary Evaluation 63
5.4.1 Implementation 63
5.4.2 Methodology 63
5.4.3 Asymmetric Synchronization 64
5.4.4 Scheduling 65

6 Conclusions and Future Work **67**

Bibliography **71**

A Correctness Proofs **79**
A.1 RR Quorum Systems 79
A.2 Distributed Register 80
A.2.1 Specification 81
A.2.2 Proof 81

A.3 State Machine Replication 82
    A.3.1 Specification 82
    A.3.2 Proof 82

List of Figures

2.1 Log-structured merge-tree (LSM-tree). A read request first consults the bloom filters (in this case, the first yields a false positive, the second a true negative and the third a true positive) and subsequently performs a binary search on the fence pointers to find the block to get in each run, finding the target key (shaded) in the third run. 14

2.2 Tiered LSM-tree. 15

2.3 Leveled LSM-tree. 15

2.4 Comparison between a tiered and leveled LSM-tree. The dashed boxes represent the capacity of the level and contiguous blocks denote sorted runs. In the tiered LSM-tree, adding another run to the first level would trigger a sort-merge which would create another run at the second level. In the leveled LSM-tree, each level is already sorted. 15

3.1 States of a node in the RR model. Compared to the crash-fault model, there is an additional "Suspicious" state. 22

3.2 Example of a partially intersecting RR quorum system ($M_R = 4$, $F = 2$). In (a), there were 3 restarts (shaded) and 2 rollbacks (crossed). Thus, the read quorum receives suspicious replies and grows larger, ensuring that both read quorums still intersect W_Q in at least one up-to-date replica (as required by **I1**). In (b), there are no restarts (and, by extension, no suspicion of rolled back state), and thus read quorums can be disjoint, as noted by **I2**. 27

3.3 Time diagram of the register write operation. 31

3.4 Time diagram of the register read operation. If the value is stable, the read can return at point A. If the the write quorum gathered in the first phase is unanimous, the read can return at point B. In this case, for clarity, the stabilization is omitted in the figure. Else, the most recent value is written back to a write quorum, and then the read operation returns at point C, issuing an asynchronous stabilization phase. 32

3.5 Pseudo-code for the register read operation. 33

3.6 Pseudo-code for the register write operation. 33

3.7 Pseudo-code for the normal case SMR operation. 34

3.8 Diagram of a normal case SMR operation 35

3.9 Time diagram of the state machine replication (SMR) read-opnly operation. If the optimized read is successful, the operation returns at point A. Otherwise, it falls back to a regular SMR operation, returning at point B. 36

3.10 Read-write register 39

3.11 State machine 39

3.12 Latency for different configurations and models in the 1ms topology. Crash Fault Tolerant (CFT) uses $F = 3$ and $RR(M_R, F, s)$ denote RR with parameters M_R and F, with s restarts. 39

3.13 Read-write register ($1ms$) 40

3.14 State machine ($1ms$) 40

3.15 Read-write register (AWS) 40

3.16 State machine (AWS) 40

3.17 Operation latency for different protocols in different network topologies 40

3.18 Read-write register 41

3.19 Replicated state machine 41

3.20 Throughput-Latency curve for different protocols 41

4.1 Slow but trivially correct mixing of the protocols. The read operation is in all aspects similar. 47

4.2 Protocol for the fast intermixing of the distributed register and SMR protocols in TEEMS. By overlapping (and piggybacking) the optimized read to the state machine to get the policy and the reading of the current timestamp of the register (which is the first step of the write protocol), we can run both operations in parallel, since writing the value to the register only happens after the policy is verified. Note that the read operation can be similarly piggybacked (the value is only returned after the policy check) 48

4.3 Deployment scenarios: (a) cloud client (with either Redis or S3), (b) collocation center with Redis. The shaded area represents a LAN (or a data center). Latencies will be used in the evaluation (in the case of S3, R represents the latency of reads and W the latency of writes). C is the client, R is a replica and P is the proxy replica. 49

4.4 Client in the Cloud w/ S3 51

4.5 Client in the Cloud w/ Redis 51

4.6 Collocation Center w/ Redis 51

4.7 Latency in different deployment scenarios. The “s3” and “redis” bars refer to direct accesses to the untrusted storage, providing a performance baseline without rollback protection. “teems” refers to accesses without caching. “teems + name cache” and “teems + blob cache” refer to accesses where the name and blob caches had a hit, respectively. Caching only applies for get requests, and in the cases of Figures 4.4 and 4.5 the “value cache” exists, but is close to zero since the cache hit with a local read quorum means there is no network latency access in the critical path. 51
4.8 Redis baseline 52
4.9 TEEMS + Redis 52
4.10 Latency with YCSB workloads 52

5.1 Architecture of the R2-S2 prototype 58
5.2 Examples of valid static schedules for a parameterization of 5 nodes with $M_R = 2$. In each schedule a line represents a replica and the temporal sequence of objects it writes (boxes). A shaded box represents an object that was synchronized immediately, while a white box represents a batched object. 59
5.3 Diagram of the Storage Layer 61
5.4 Maximum throughput for different configurations. The batch and reply configuration places the object in a batch and replies immediately, while in the sync immediately configuration all replicas flush the object to the persistent log before replying. In the $T(BR, 5)$, BR replicas in each write use the former approach, while the remaining use the latter. 64
5.5 Maximum throughput for different schedules. The batch and reply configuration places the object in a batch and replies immediately, while in the sync immediately configuration all replicas flush the object to the persistent log before replying. The constant schedule always synchronizes to the same 3 of the 5 replicas. The randomized schedule also synchronizes to 3 replicas, but these get randomized every operation. The round robin schedule changes the 3 replicas at each operation in a cycle. 65

List of Tables

2.1 Quorum sizes in different fault models. In the crash model f is the number of crashed replicas, in the Byzantine model f is the number of Byzantine replicas, and in the UpRight model u is the number of crash faults and r the number of commission faults tolerated. The RR quorum sizes are also included as a comparison point, where F is the number of crash faults, M_R is the maximum number of rollbacks and s is a runtime value corresponding to the number of replicas which are suspicious of their state (further details can be found in Chapter 3) . . . 10

3.1 Parameters required by different fault models. . . . 40

4.1 TEEMS Interface . . . 46

5.1 Availability and reliability in a geo-replicated deployment (uncorrelated failures) versus a data-center deployment (correlated failures). R_Q and W_Q are the sizes of the quorums for a parameterization with M_R tolerated rollbacks and F crash faults . . . 62

Acronyms

ABD	Attiya, Bar-Noy, Dolev
BFT	Byzantine Fault Tolerant
CFT	Crash Fault Tolerant
KVS	Key-Value Store
LSM-tree	Log-structured merge-tree
TEE	Trusted Execution Environment
OS	Operating System
R2-S2	Restart-Rollback Storage System
RR	Restart-Rollback
SMR	state machine replication
TEE	Trusted Execution Environment
TEEMS	TEE Metadata System
VFT	Visigoth Fault Tolerant
YCSB	Yahoo! Cloud Serving Benchmark

1

Introduction

Contents

1.1 Contributions 5
1.2 Organization of the Document 6
1.3 Publication 6

Replication is a standard technique in distributed systems, which adds fault tolerance to a service implemented by an individual networked node. To ensure that the replicated service appears to clients like its single-node equivalent except for higher availability, a replication protocol coordinates the replicated nodes. For instance, to replicate a Key-Value Store (KVS) (i.e., a storage system which offers a simple get/put interface) one can use the Attiya, Bar-Noy, Dolev (ABD) [1] protocol. For more complex applications that can be modeled using a deterministic state machine, a state machine replication (SMR) protocol like Paxos [2] may be used.

Durability, i.e. not losing data even in the event of faults, is a key property of storage systems. For replicated storage systems, adding local persistence to the nodes that make up the system is a common practice. This allows replicas to recover from crashes (in which they lose their volatile state) from their local persistent state, which in turn makes the system resilient to catastrophic full system shutdowns, a property considered as being of paramount importance in real-world deployment of replicated systems [3]. From a formal point of view, however, replicated systems that persist the state of each replica durably are commonly treated in a very similar fashion to others that store their state solely in volatile memory — the persistent aspect is considered trivial. Indeed, when the persistent aspect of nodes is explicitely highlighted, protocols only add small extensions that describe when and what data nodes need to persist so that, in the event of a restart, the node can operate as if nothing happened.

This assumption that persisting the local state of replicated nodes is trivial does not hold in certain security sensitive cases, in particular in the context of Trusted Execution Environments (TEEs). TEEs are applications running in a special hardware-protected environment, where the CPU shields the application, guaranteeing the integrity and confidentiality of the code execution and the associated volatile memory, in a way that it can be remotely attested that the hardware protections were correctly set up. Consequently, they have become an attractive solution for system administrators who make deployments in infrastructure that they do not directly control (e.g., the public cloud). TEEs run in an highly adversarial attacker model, where the owner and operator of the hardware is not trusted. As such, although nodes can keep integrity information (i.e., a cryptographic hash) in memory and encrypt their persistent state, to ensure that it cannot be arbitrarily tampered with, their control over the persistent storage is quite limited. Particularly, when a restart happens and the volatile integrity information is lost, the TEE may suffer a rollback attack, where the attacker replaces the persistent state with a stale but correct version of the state. Simply applying current replicated protocols would pose a correctness and security problem.

Despite these challenges, networked services implemented inside TEEs like ARM TrustZone [4], AMD SEV-SNP [5, 6], Intel SGX [7], or the future Intel TDX [8] and ARM CCA [9] can similarly benefit from replication. Replicated TEE-based systems aim to combine the confidentiality, integrity, and remote attestation of TEEs with replication to ensure high availability. Examples include permissioned blockchains [10], monotonic counters for ensuring state freshness [11], in-memory KVSs [12], as well as broadcast and common random number generator primitives [13].

A key design decision in all replicated systems is the choice of fault model underlying the replication protocol. This model captures the set of faults that can affect an individual replica. For instance, in the crash-fault model, faulty nodes are assumed to execute correctly until they crash at an arbitrary point in their execution, when they seize to perform further steps until they restart. In the Byzantine fault model, faulty nodes may perform arbitrary actions. The choice of fault model affects the complexity, overhead, and tolerance threshold of a replication protocol, and is important for both performance and security. A fault model that is too pessimistic may lead to unnecessary safeguards, which can impact both performance and the cost of replication. Conversely, a fault model that is too optimistic may fail to account for all faults that can occur, which then leads to broken assumptions and loss of correctness or security of the replicated system.

In the case of TEE-based replication, the choice of existing systems that store replica state persistently [10, 11] was to employ Byzantine Fault Tolerant (BFT) replication. This may seem necessary given that rollback of persistent state is a behavior not covered by the crash-fault model. However, the Byzantine model is much stronger than necessary for this setting. Intuitively, this is because it assumes arbitrary behavior of faulty components, thus not taking into account the integrity guarantees for the code running inside TEEs.

To fill this gap in distributed replication research, we introduce the Restart-Rollback (RR) fault model, which captures precisely the set of faults that TEEs can suffer according to their specification. The main insight behind RR is that, apart from crash faults, whenever a TEE restarts, the fault model allows for a rollback of externally stored state to an earlier version. Such rollback events are possible because TEEs lack general and high-performance means of ensuring the freshness of externally stored state. During an execution, however, the integrity of the internal, volatile state of a TEE is ensured by hardware, and the integrity of any externally stored state can be ensured by standard cryptographic means.

The RR fault model occupies a middle ground between Crash Fault Tolerant (CFT) and BFT models. As we will show, unlike CFT it provides security for replicated TEEs by tolerating state roll-back at a cost that is similar to CFT and better than BFT. Specifically, it is able to substantially reduce the required communication, particularly for read operations, in the common case where restarts are infrequent.

Stepping back, we make the observation that the RR fault model can be useful, albeit for different reasons, in the more general case of persistent replicated systems within the crash model. A key factor for performance in this systems are the slow accesses to untrusted storage, which must occur in the critical path to ensure the correctness of the system. By batching multiple requests together, it is possible to reduce this overhead, increasing throughput at the expense of the additional latency required to form the batch. Leveraging the RR model, it is possible to form larger batches, replying to certain requests before synchronizing to persistent storage. This would otherwise be unsafe, as a crash before the synchronization would constitute an effective rollback, which the RR model captures.

To demonstrate the potential of the RR model, we apply it to two types of replication protocols, namely the ABD read/write replication protocol and the Paxos SMR protocol, showing that this adaptation is relatively straightforward. We then use these protocols to devise two systems: TEE Metadata System (TEEMS) and Restart-Rollback Storage System (R2-S2). TEEMS, a replicated metadata service, is a relevant contribution in its own right, since it enables Cloud storage with the same strong confidentiality and integrity guarantees provided by TEEs. while also ensuring freshness of data. TEEMS is used with one or more untrusted cloud storage services. TEEMS maintains metadata for each data item (including the encryption key, authentication code, latest version number and policy), while ensuring strong security and enabling concurrent sharing of data. R2-S2 is a replicated KVS built using regular nodes (i.e., without TEEs or other security mechanisms) that leverages the RR model with the objective of extracting higher throughput while retaining the strong reliability expected of a persistent replicated system.

1.1 Contributions

In the TEE context, we implemented the read/write register and the SMR protocols in three fault models: CFT, RR and BFT, and a prototype for TEEMS, all using Intel SGX. In the storage context, we implemented R2-S2 as a replication layer on top of a single node key value store. We evaluated our prototypes using both microbenchmarks and real-world workloads, namely Yahoo! Cloud Serving Benchmark (YCSB) [14].

Our contributions include:

1. The RR fault model, which is the first to capture precisely the fault behavior of TEEs that rely on external storage;
2. A derivation of quorum sizes for replication protocols in the RR model;
3. Principles for the adaptation of CFT protocols to the RR model;
4. Two protocols for this model, implementing read-write storage and SMR;
5. The TEEMS generic metadata service, and its integration with untrusted cloud storage services to give clients the ability to share and concurrently access persistent data with strong confidentiality, integrity, and freshness;
6. The R2-S2 distributed KVS, which enables higher throughput from KVSs backed by persistent state;

7. An extensive experimental evaluation of our protocols in the TEE setting, with a detailed comparison with unsafe CFT and expensive BFT solutions;
8. An preliminary experimental evaluation of the R2-S2 system.

1.2 Organization of the Document

The rest of the thesis is organized as follows: Chapter 2 covers related work. In chapter 3 we formalize the RR model and show the derivation of RR-based quorum systems and protocols. Chapter 4 describes the usage of TEEs with RR, with the accompanying design and evaluation of TEEMS. In chapter 5 we present the design of R2-S2 and its experimental evaluation. Chapter 6 summarizes our work and presents possible avenues for future work.

1.3 Publication

A portion of this thesis, namely most of the material covered in Chapters 3 and 4, is featured in a paper accepted to the Network and Distributed System Security (NDSS) Symposium 2023.

2

Related Work

Contents

2.1 Fault Models 9
2.2 Replication Protocols 11
2.3 Trusted Execution Environments 11
2.4 Single-node Key-Value Stores (KVSs) 14
2.5 Persistence in replicated distributed systems 15

2.1 Fault Models

In this section, we survey the most used classes of fault models for distributed systems and compare them to the novel RR model. It is important to first clarify that, both here and in the rest of this document, we consider the network to be asynchronous. This means that the upper bound for the message delivery delay between replicas is unknown, and may not even exist. This model is consistent with the Internet, where network partitions may temporarily impede or arbitrarily slow down the communication between two nodes.

Crash fault models. A crash fault is said to have occurred when a node stops replying to requests. This may happen for several reasons: power outages, network partitions, programming errors that cause the processes to crash, etc. A crash fault can be considered fail-stop if other processes can detect that it has in fact crashed. However, perfect failure detection in asynchronous systems is impossible [15]. A node that suffers a crash fault is indistinguishable from an extremely slow one [16] (rigourously, a node which replies at $t = \infty$). The crash-recovery model [17, 18] is a slight variation from the crash model, wherein replicas are allowed to restart and run a recovery protocol.

Byzantine fault models. First introduced by Lamport et al in [19], a Byzantine fault is the most encompassing type of fault possible, since it assumes nodes can deviate arbitrarily from the prescribed protocol. They may do so maliciously or simply by component or programming error. Furthermore, several Byzantine nodes may collude to deviate in a coordinated fashion, in an attempt to subvert the guarantees of the system. Compared to crash fault tolerant protocols, BFT protocols are expensive on account of mechanisms such as digital signatures [20] or extra protocol rounds [21].

Hybrid fault models. Hybrid fault models consider at least two types of faults separately. They have the advantage of better adapting the system parameterization to the deployment conditions by offering greater flexibility, at the cost of some complexity. UpRight [22] is an example of a system which uses such a hybrid fault model, concretely a hybrid crash-BFT model, where the system is able to sustain a different number of crash and Byzantine faults. In particular, UpRight sustains u crash faults in addition to r commission faults, which are a subset of Byzantine faults where nodes actively produce an incorrect message (and thus excluding the subset of crash faults from the Byzantine failure model). This leads to an elegant reasoning about Byzantine fault tolerance that separates incorrect from non-incorrect node behavior.

Besides protocol changes, different fault models require different quorum systems. A quorum is defined as the minimum set of replies that need to be gathered to proceed to the next step of the protocol. The size of quorums is of great importance for the economical cost of systems, as larger quorums require more replicas and resources, as well as their performance, as larger quorums will generally take longer to gather. These quorums not only depend on the fault model, but on the network model and synchrony assumptions. As an example, Table 2.1 has the quorum sizes for the crash, Byzantine and UpRight models in the asynchronous model.

Table 2.1: Quorum sizes in different fault models. In the crash model f is the number of crashed replicas, in the Byzantine model f is the number of Byzantine replicas, and in the UpRight model u is the number of crash faults and r the number of commission faults tolerated. The RR quorum sizes are also included as a comparison point, where F is the number of crash faults, M_R is the maximum number of rollbacks and s is a runtime value corresponding to the number of replicas which are suspicious of their state (further details can be found in Chapter 3)

Fault Model		Fault Parameter	# Replicas	Quorum Size
Crash		f	$2f+1$	$f+1$
Byzantine		f	$3f+1$	$2f+1$
UpRight		u, r	$2u+r+1$	$u+r+1$
Restart-Rollback	Read	M_R, F	$\max(M_R, F)+F+1$	$\min(M_R, s)+F+1$
	Write			$\max(M_R, F)+1$

Comparison to RR. The RR model allows for the system to tolerate a combination of crash faults and faults where a replica has stale state after a restart. The RR model is strictly more encompassing than the crash model, allowing for fault behaviors that are not tolerated by this. Compared to BFT, it assumes the correctness of the implementation of the protocol by all replicas, and as such does not need to protect against generic Byzantine faults, leading to simpler, more efficient protocols with smaller quorums in the common case. Finally, compared to hybrid crash-BFT models [22–25], while RR is also a hybrid model, it differs in the type of non-crash behavior it admits, since wrong answers in RR are limited to rollbacks after a restart, whereas previous hybrid models tolerated arbitrary Byzantine faults.

The RR model has some relevant characteristics such as leading to dynamic quorum sizes, where the size depends on the number of replies from recently restarted nodes, leading to small read quorums in the normal case. Other proposals, such as Refined Quorum Systems [26] or Flexible Paxos [27], also use varying quorum sizes in different parts of the protocol. However, they do it in a way that is complementary to ours, since they work on traditional models (crash and Byzantine) and come up with protocol techniques based on different quorum sizes to improve performance. In contrast, RR is a new model (based on a different set of assumptions) that naturally leads to dynamic quorums.

Another approach of hybrid quorums has been put forward by Visigoth Fault Tolerant (VFT) [28]. In VFT not only are crash and commission faults separately considered, the synchrony of the system is also a parameter of the quorum system. This allows VFT to be parameterized as either synchronous or asynchronous and as either crash or Byzantine, with hybrid combinations. Although an interesting contribution, we consider VFT to be an orthogonal approach.

2.2 Replication Protocols

Replication is a technique wherein multiple copies of a service run simultaneously so that the overall system maintains its operation even in the presence of faults. There are several protocols for replicated services, varying vastly according to the types of faults tolerated, performance characteristics and semantics of the service. In this section, we will survey two types of services frequently replicated: read-write registers and state machines.

Read-Write Register. A read-write register supports a very simple interface, allowing clients to either read the current value or writing a new one. Due to their simplicity, protocols for replicating registers are considered to be quite fast. The ABD protocol proposed by Attiya et al. [1] implements this register in the crash model, while the Malkhi and Reiter proposed a Byzantine fault tolerant counterpart in [20].

State Machine Replication. A deterministic state machine (SM) is a mathematical construct that supports the successive application of operations, which mutate its internal state. This is a powerful programming abstraction, as it can simulate any deterministic computation. For that reason, it has been the object of much research within the distributed systems community: using state machine replication (SMR), system builders can build arbitrarily complex fault tolerant systems [29] (provided that they are deterministic). A common method for achieving SMR is to attach a consensus module to an operation log. Consensus is a process by which a set of nodes agree on a value. By agreeing on the value for each entry in the log, replicas can then be sure that they apply the same operations to their local state machines in the same order, guaranteeing that their state is consistent. Importantly, Fischer, Lynch and Paterson proved that consensus is impossible in a fully asynchronous network [30]. As a result, consensus protocols make synchrony assumptions in certain places.

The Paxos protocol [2] proposed by Leslie Lamport, which is applicable to the crash model, has been widely studied, with multiple variants being proposed [27, 31–34]. Raft [35] is an alternative protocol for SMR in the crash model, developed with the goal of being easy to understand and implement. The PBFT [21] protocol developed by Castro and Liskov provides SMR in the Byzantine model.

2.3 Trusted Execution Environments

Recent generations of general-purpose CPUs, namely the ones that support Intel SGX [7], Intel TDX [8], AMD SEV-SNP [5, 6], or ARM TrustZone [4] technology, include support for trusted execution environments (TEEs), allowing users to deploy a computation with hardware-enforced confidentiality and integrity on a compute platform that is not under their control. TEEs have been used to protect the integrity and confidentiality of generic applications running on commodity operating systems [36], as well as of large-scale distributed computations [37]. Microsoft and Google both offer VMs with TEE support as part of their Cloud services [38, 39].

Built-in storage support for TEEs. Current TEEs offer some support for securely storing persistent data in an encrypted form outside the TEE, such that it can be decrypted only by the TEE that stored it if and only if it was not modified. If this support would offer perfect protection, then a CFT replication protocol might suffice to replicate TEE-based systems. However, different systems have different limitations that preclude a direct use of CFT. In particular, ARM TrustZone-based devices include an eMMC storage controller that allows for configuring a small storage partition (on the order of tens of megabytes), named the “Replay Protected Memory Block” (RPMB), which only accepts commands from authenticated secure world domains, and withstands rollback attacks [40]. However, the size and access to this block is quite inflexible: the RPMB is configured as a separate partition of the eMMC storage device, and a shared cryptographic secret must be embedded in the host and the storage device during manufacture. Intel SGX supports data sealing, i.e., encrypting it in a way that it can be decrypted only by the same enclave on the same platform [7]. However, this abstraction alone does not protect against rollback attacks, where an attacker replaces the most recent sealed data object stored in an untrusted medium with a stale version of the object [41]. Recent versions of SGX use a persistent, monotonic counter that is uniquely tied to the TEE instance by packaging the counter storage physically with a CPU, which is itself tied to the TEE [7]. Upon sealing, this counter is read from the CPU, incremented and stored with the sealed data. Upon unsealing, the counter that was previously stored with the data is compared to the current counter value in the CPU. However, the authors of a recent paper [11] note that writes to the Intel SGX monotonic counter are very slow (on the order of hundreds of milliseconds) and that wear can cause its memory to fail after a few days of continuous use.

Protection against rollback attacks. The storage associated with the TEE can also be secured using methods other than the built-in support. The strategies used by these methods fall into two categories. The first category of defenses relies on a small amount of non-volatile state tied to the TEE platform, which is used to assert the freshness of the TEE’s externally stored state upon a restart of the TEE. Memoir [41] relies on a TPM [42] to store a hash chain of all external state updates. To avoid wearing out the TPM’s NVRAM, whenever the TEE executes an operation, it extends the hash chain in a volatile TPM PCR register. Upon power failure, an uninterruptible power supply is used to run a shutdown handler that copies the latest PCR value into the TPM’s NVRAM. ICE [43] relies on a specialized CPU extension with volatile “guarded memory”, which stores the latest freshness information. During a power failure, a capacitor supplies enough power for the CPU to write the freshness information into off-chip, untrusted NVRAM. Ariadne [44] writes freshness information to on-chip NVRAM, but minimizes wear by using a Gray code to represent a monotonic counter, such that any increment requires only a single bit flip. Ariadne also describes an attack with the “store-then-increment” monotonic counter approach where a TEE is made to crash repeatedly between the store and the counter increment, and proposes a fix by incrementing the counter twice upon a recovery from a crash.

The drawback of defenses in this category is that, in practice, either the rate at which a TEE can save its state is limited by wear of the non-volatile memory (latest SGX, Ariadne), or there is a dependency on specialized hardware or an uninterruptible power supply (ICE, Memoir). A second category of defenses stores freshness information in a separate trusted server, or set of servers that are assumed to not all be subject to a coordinated rollback attack.

An approach called ROTE [11] addresses the performance and wear shortcomings of monotonic counters in Intel SGX by replicating a monotonic counter in several TEEs, using what is a hybrid crash-BFT protocol [22, 23], where a subset of the replicas may become unavailable and another subset may return arbitrary results. However, since ROTE maintains the counter in the enclave’s volatile memory, it will lose the counter state after a simultaneous restart of all replicas, e.g., due to a system-wide power failure. Moreover, the use of BFT is too pessimistic and leads to costly replication and poor performance, as we discussed.

Defenses in both categories, although adequate for single-client applications, fundamentally cannot support concurrent sharing between clients since they do not provide atomic updates of data and metadata.

Protection against forking attacks. A forking attack on a TEE occurs when two instances of the same enclave are created and executed in parallel, unbeknownst to each other. Such an attack can create a branch in the state of the TEE, breaking state continuity. To circumvent these issues, there are three main approaches, either relying on trusted monotonic counters [11, 41, 44–46], on client cooperation/intervention [47–53] or on some centralized (possibly replicated) source of truth [36]. Relying on monotonic counters shares the disadvantages already discussed, and relying on clients may be impractical or even impossible (as it requires mutual trust between clients or they being online). In the context of replicated systems, using the system configuration (modified using consensus) as the source of truth is a natural mechanism to prevent forking attacks.

Replication using TEEs. There are some proposals that, although not focusing specifically on the problem of persistence, share the goal of improving replication protocols through trusted components. These can be broadly divided into proposals that run an entire replica inside a trusted component [13, 54] and those that augment the replica logic with a small trusted module [45, 46]. However, their technical solutions are inherently different from the ones we adopt, because they either trust the entire replica or have a trusted computing base that is constrained by the hardware (e.g., a trusted monotonic counter).

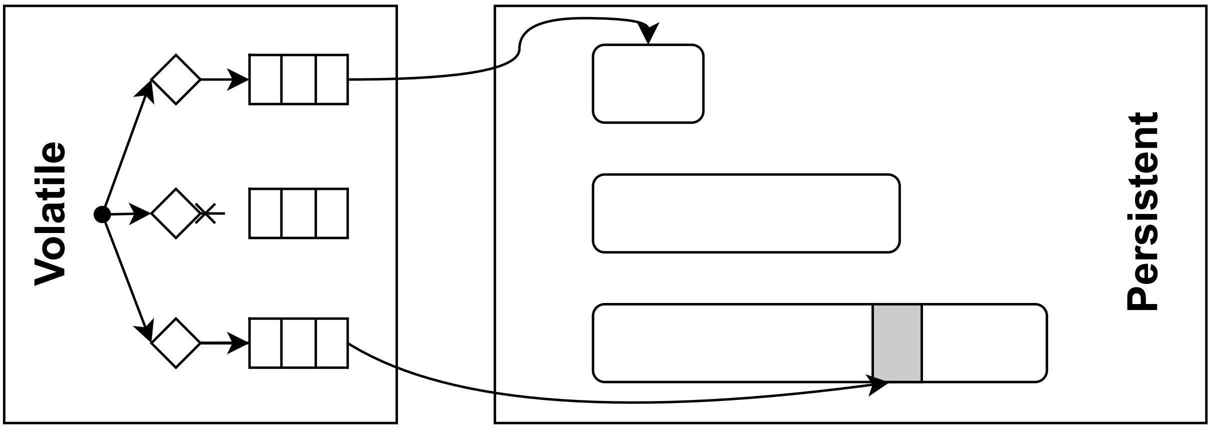


Figure 2.1: Log-structured merge-tree (LSM-tree). A read request first consults the bloom filters (in this case, the first yields a false positive, the second a true negative and the third a true positive) and subsequently performs a binary search on the fence pointers to find the block to get in each run, finding the target key (shaded) in the third run.

In contrast, our work focuses on a setting where the computation is trusted but the persistent storage is not.

2.4 Single-node Key-Value Stores (KVSs)

One way of implementing replicated KVSs is to implement a replication layer on top of a single-node KVS. Consequently, it is important to understand the overall design of a single-node KVS, as well as its tradeoffs.

2.4.1 Log-structured merge-trees (LSM-trees)

The key data structure for modern KVSs is the Log-structured merge-tree (LSM-tree). Examples of industry-grade KVSs relying on LSM-trees include LevelDB [55], BigTable [56], Apache Cassandra [57] and RocksDB [58]. LSM-trees have also been the topic of a lot of recent research [59–62].

The main objective of LSM-trees is to improve the scalability of both reads and writes of persistent KVSs. To scale writes, the LSM-tree creates an in-memory batch of objects. Once it has filled up, it sorts the batch and sends it to disk where it becomes a run. To keep the data organized for efficient reads, the LSM-tree maintains similarly sized runs on levels of exponentially increasing sorted runs. Once a particular level becomes too full, it is sort-merged into a run of the following level. To perform a read, a binary search is performed on each run to find the object. To optimize IO costs, two approaches are employed. First, arrays of fence pointers are stored in memory, which allows us to perform the binary

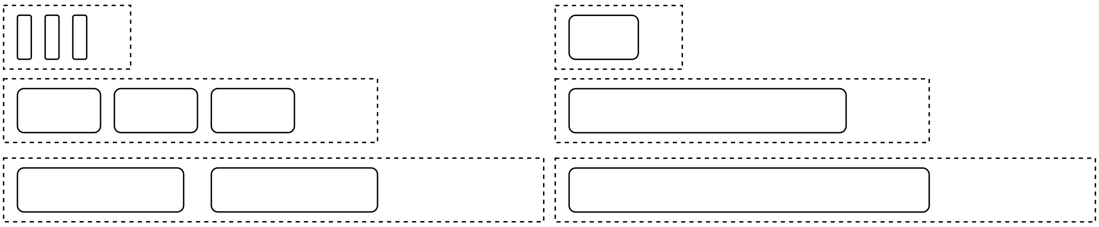


Figure 2.2: Tiered LSM-tree.

Figure 2.3: Leveled LSM-tree.

Figure 2.4: Comparison between a tiered and leveled LSM-tree. The dashed boxes represent the capacity of the level and contiguous blocks denote sorted runs. In the tiered LSM-tree, adding another run to the first level would trigger a sort-merge which would create another run at the second level. In the leveled LSM-tree, each level is already sorted.

searches in memory, reducing the IO cost per run to 1. Additionally, a Bloom filter [63] can be constructed per run to filter upfront which runs need not be checked. Figure 2.1 summarizes the architecture of an LSM-tree responding to a read request.

There are two classes of designs for LSM-trees, levelling and tiering LSM-trees. In the tiereing approach, each level gathers runs from the previous level, sorting them only when it reaches capacity and pushing the newly-sorted run to the next level. In the levelling approach, a run from the previous level is merged as soon as it is received, ensuring the run is always sorted. The tiering approach is tailored to write-intensive workloads, since it does less work per object, while the levelling approach is more read-optimized, as the runs are in a sorted state, allowing for more efficient lookups. Figures 2.2 and 2.3 show the two different approaches.

2.5 Persistence in replicated distributed systems

The means by which persistence is achieved in replicated systems is non-consensual within the distributed systems community. Some authors consider that persistence requires synchronizing the state of the replicas to a persistence medium [64] (be it a magnetic disk, a solid state drive or non-volatile memory) while others argue that persistence can be achieved by replication [21, 65–67]. The latter approach, while avoiding expensive accesses to persistent storage, makes it impossible to sustain a full system shutdown. In practice, catastrophic failures caused by natural disasters are generally unavoidable. Moreover, forcing some replicas to be always up makes it difficult to upgrade replicas simultaneously.

Persistence in distributed read-write registers Read-write register protocols have clear places where the state needs to be persisted in storage: when a replica receives a write (or writeback) request. If the intended system is to be fully in-memory, then this synchronization can be skipped. In general, protocols for implementing these registers omit this discussion [1, 68], leaving the decision for system designers.

Persistence in replicated state machines Replicated state machines have been extensively studied from the point of view of persistence and recovery. Although it is dubious whether replicas are expected to synchronize to persistent state in the original formulation of the Paxos algorithm [2], the clarification paper [69] published by Lamport specifically requires replicas to synchronize to persistent storage before replying.

Nevertheless, this has not prevented further research from deviating from this prescription. PBFT [21], which implements replicated state machines in the Byzantine model, considers that persistence is achieved through replication. Even in the crash model, there have been attempts to either remove or reduce the impact of persistent storage. Boichat et al. [70] describe a recovery protocol for Paxos, which they call Winter, where persistent storage is never used. However, this is done by assuming that a strict majority of nodes is always up, which severely limits the practicality of the solution.

A more recent attempt [71] to improve on recovery protocols for replicated state machines using Paxos proposed two protocols which only write to disk infrequently (either on leader changes or in the case of a restart). They achieve this by assuming that a majority of nodes is never simultaneously faulty. This assumption, albeit stricter than the one made by Winter, still makes it impossible to sustain a full system shutdown.

3

The RR Fault Model and Its Protocols

Contents

3.1 Motivation 19

3.2 RR model definition 22

3.3 Replication protocols 29

3.4 Evaluation of TEE replication protocols 37

3.5 Discussion: Generalization of the RR model 42

In this chapter, we define the RR fault model, which captures the behaviour of replicated nodes with persistent state.

We will motivate this fault model from its origins as a fault model for replicated TEEs with persistent state. Subsequently show how this fault model can also be applied in the context of replicated storage systems with persistent storage that operate outside of the adversarial model of TEEs, where CFT is sufficient for correctness but RR can be leveraged for performance. Then, we derive requirements for quorum systems in the RR model, as well as their intersection properties, a key element to the correctness of replication protocols.

Building on the quorum system abstraction, we explain how to adapt existing CFT protocols to the RR model. We provide two concrete examples of such adaptations using well-established protocols: the ABD [1] protocol for a read-write register and the Paxos [2] protocol for SMR. We then provide a performance evaluation in the context of TEEs using micro-benchmarks. We compare the CFT protocols with their adaptations to the RR model and the equivalent counterparts in the BFT world. This demonstrates the viability and usefulness of the RR model in the context of TEE-based replication, showcasing how it enables protocols with both correctness (otherwise only found in BFT) and speed (present in CFT-based replication). We conclude this chapter with a discussion.

3.1 Motivation

3.1.1 TEE properties

In this section, we review the properties of TEEs and motivate the RR fault model. While the precise guarantees provided by a TEE vary with the TEE design, we can summarize common guarantees across platforms.

Confidentiality. TEEs allow a computation to execute with hardware-enforced confidentiality over the internal code and data used by the computation. Data and instructions are decrypted in hardware as they are fetched from memory inside the CPU chip and modified data is re-encrypted before it leaves the CPU chip. Only code executing inside the TEE has access to cleartext data, therefore ensuring confidentiality even from Operating System (OS), hypervisor, and platform operators.

Integrity and attestation. When a TEE is started, the secure platform computes a hash of the TEE's initial code and data and compares it to the expected measurement hash for the instance. Only when this attestation succeeds is the TEE provided with the key material it needs to authenticate itself to third parties and to access and decrypt persistent data stored externally on its behalf. A remote party can ascertain that it is communicating with a TEE instance that has a particular initial measurement hash and executes on a legitimate TEE platform via remote attestation. Furthermore, to protect the TEE's integrity during execution, the hardware isolates the TEE and detects modifications of encrypted code and data while stored in main memory. Some implementations like Intel SGX even protect code and data from certain physical attacks.

State continuity in the presence of external state. TEEs can stored encrypted state in external persistent storage across activations through a process called sealing. As described above, a correctly attested TEE receives a secret key unique to its instance, which allows the TEE to store encrypted external state with confidentiality and integrity guarantees. To ensure the recency of its external state, however, a TEE must ensure that the encrypted external state it is presented with after a restart is the most recent version it had previously stored. More generally, TEE computations may require the strictly stronger property of state continuity, which requires that a TEE never executes an operation with a stale state, or executes different inputs from the same state [44].

TEE implementations lack general, high-performance support to ensure freshness and state continuity for computations with external state. Some TEE platforms provide trusted, persistent monotonic counters associated with the CPU platform. While these counters are sufficient in principle to ensure state continuity, they are not sufficient in practice [44]. In particular, hardware intentionally slows the time to increment these counters to milliseconds in order to avoid wrap-around attacks [11, 44]. As a result, such counters can at best support state continuity for TEEs that exhibit infrequent, orderly shutdowns, during which a TEE can update the counter and store its external state with the latest counter value embedded [44]. However, trusted counters are inadequate for TEEs that frequently update their external state (e.g., a database or KVS) and can crash at any time; ensuring state continuity for such TEEs requires rapid counter updates. Moreover, trusted counters are tied to a particular CPU/motherboard and do not support safe migration of TEE computations.

Note that state continuity implies fork protection [50], i.e., guaranteeing that no duplicate TEE are instantiated with access to the same stored state. We discussed how fork protection and how it can be achieved in replicated systems in Chapter 2.

TEE threat model and guarantees. The design of TEEs assumes a powerful adversary, who has full control over the operating system and hypervisor that hosts a TEE. The adversary can arbitrarily create and shutdown TEE instances at any time, as well as delay, read, drop or modify all messages sent to and received by enclaves. Moreover, an adversary can tamper with or replace the external (encrytped) state associated with a TEE instance.

TEE security is rooted in the hardware design and implementation, as well as the vendor's certificate chain used for remote attestation. As a result, the threat model of TEEs excludes compromise of the vendor's TEE platform design and implementation, physical attacks on the CPU chip, or compromise of the vendor's certificate chain. Some TEE implementations also exclude physical attacks using bus probes. Side channels are outside the threat model of current TEEs.

Choosing a fault model for replicated TEEs Subject to the TEE threat model, computations that do not depend on external state enjoy confidentiality and integrity, and can be considered to suffer only crash faults if the codebase is trusted (as opposed to Byzantine faults, where an adversary may induce arbitrary behavior in a component). Therefore, a crash-tolerant replication protocol is sufficient to replicate TEE-encapsulated computations that don't use external state. If a TEE computation relies on external state, however, then it can additionally suffer a rollback of the external state to an earlier version whenever the TEE restarts. This behavior is beyond the crash fault model; therefore the use of crash-tolerant replication protocols is not safe. Currently, BFT protocols are typically used instead [10, 11]. Using BFT is safe but needlessly expensive, because these protocols are designed to tolerate arbitrary behavior, most of which is masked by the properties of TEEs. In the Section 3.2, we describe a novel fault model that captures precisely the set of behaviors exhibited by TEEs with external state: crash failures plus state rollback after a restart.

3.1.2 Extending the usefulness of fault model

Although the TEE use-case requires something like the RR model out of necessity, we also look at RR as an opportunity. The ability to tolerate some rollbacks on restart is useful outside the security sensitive context of TEEs. Consider a replicated KVS, where each node maintains a local KVS. When writing to the local key value store, there are three possible approaches:

A. The node synchronizes the written object immediately, and then replies. This impacts throughput;

B. The node places the object in a batch, and waits for the batch to be flushed before replying. This impacts latency;

C. The node places the object in a batch, and immediately issues the reply. This offers the best throughput and latency, but can lead to data loss if the replica crashes (i.e., if it suffers a rollback);

RR opens the possibility of a hybrid approach, where some nodes place the object in the batch and reply eagerly (approach C), while others bypass the batch and synchronize directly to disk (approach A). With careful orchestration, this can in principle allow for a throughput advantage compared to approach A, a latency advantage compared to approach B and a durability advantage compared to approach C, a very interesting tradeoff.

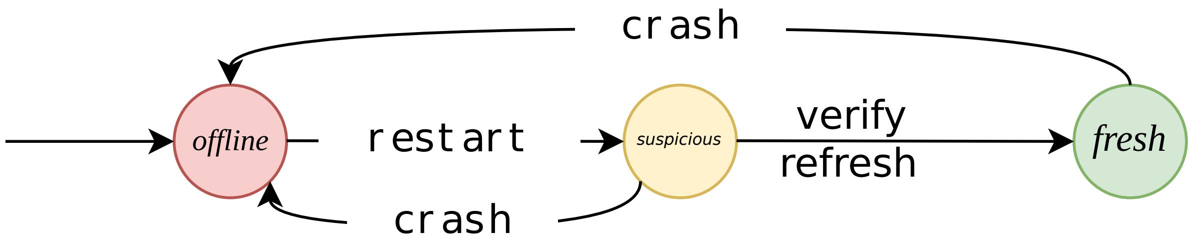


Figure 3.1: States of a node in the RR model. Compared to the crash-fault model, there is an additional “Suspicious” state.

3.2 RR model definition

Nodes in the RR model are network-connected processes with external persistent state. Nodes can crash at any point in their execution and restart at a later instant. Additionally, nodes can suffer a rollback failure upon a restart, after which their externally stored, persistent state may be valid but stale. After each restart, nodes flag their state as suspicious when replying to requests, signaling that they have restarted and thus may have been subject to a rollback. Nodes can reliably determine when it has executed its initialization code and thus restarted. A node stops indicating the suspicious flag once it is ascertained that its state is fresh, e.g., by ensuring that a sufficiently large number of other nodes have the same state (as we exemplify in Section 3.3).

Nodes execute the algorithm correctly, with this exception concerning state freshness. Figure 3.1 illustrates the state transitions a node in the RR model can go through.

3.2.1 Replicated systems

Replication protocols make a separation between clients machines and replicas, where these two groups of nodes are connected by a network through which any pair of nodes may communicate, defining what is referred to as a message-passage system. These replicas collectively implement a replicated service exporting a set of operations, which clients may invoke. For instance, a storage system may export a simple interface with read and write operations, whereas a replicated database has a richer interface supporting SQL operations. To collectively implement a replicated service, replicas store their view of the current state of the system that is shared among multiple clients, and client operations are implemented by contacting a group of replicas to either query or update that state. In general, replication protocols leverage a “quorum RPC” communication pattern [72, 73] where a machine (either a client or one of the replicas) sends a message to a group of replicas and waits for a reply from a quorum. In the fault tolerance literature, a quorum is a set of subsets of the group of replicas, where these subsets have certain intersection properties that are important for the protocol correctness [74]. These intersection properties depend on the fault model, e.g., with crash faults it normally suffices that any two quorums have a non-empty intersection — majorities are an example of a quorum system with this property. Byzantine quorums, in turn, require a larger intersection to account for incorrect replies from malicious replicas [72].

3.2.2 Replication in the RR model

A key insight of replication in the RR model is to grow quorums dynamically when replicas are suspicious of their state. We quantify this suspicion by counting, in each instance of the quorum RPC pattern, how many replicas indicated the suspicious flag, and we refer to this count as a per-RPC variable s. In addition to this (dynamic) value, we also define a (static) maximum bound on the number of nodes that may actually suffer a rollback attack within the replica group, M_R.

Note that s and M_R are different and unrelated in several respects. First, s is measured by a node that gathers a set of replies, and therefore its value is specific to each invocation of an operation on a group of replicas, whereas M_R is a constant bound that is assumed to hold for that group of replicas during the entire execution of the system. As a consequence, s cannot be set at system configuration time, whereas M_R needs to be set by an administrator according to an expectation of the deployment conditions. Second, s can vary from zero (common case, no recent replica restart) up to N (simultaneous system shutdown followed by a restart). In contrast, M_R will be parameterized according to the likelihood of a correlated rollback attack, which in turn depends on the deployment and its independence expectations. For example, one could deploy replicas across different administrative domains, in which case (and assuming that there is no collusion between administrators of different domains), M_R should be at least the maximum number of replicas within a single domain. This encompasses the worst non-colluding attack where a malicious administrator rolls back all the replicas in the domain simultaneously.

Note that the parameter M_R also subsumes permanent crash faults (e.g., due to permanent hardware failure), since permanent crashes can be seen as a particular instance of a rollback, where we replace a failed node with a new one that starts from the initial state of the system (or fetches a recent but possibly not the most recent one). Finally, we define a liveness bound F, i.e. the system is live provided that no more than F replicas are temporarily unreachable at any given moment, due to network partitions, power outages, or reboot.

3.2.3 Deriving RR Quorums

As we explained, the correctness of replication protocols hinges on the property of quorum intersection: any pair of operations executed in the system must execute in replica subsets (or quorums) that intersect sufficiently for the system to return a result that obeys the protocol specification. We now revisit this intersection property for the design of protocols for the RR model. To this end, we need to first specify the set of properties that this intersection should achieve.

Property 1 (Freshness)**.** *The safety property of replicated systems normally includes the need for the most recently written value to be seen by subsequent operations. In quorum-based protocols, this property is enforced by ensuring that any pair of quorums intersects in at least one replica that does not deviate from its prescribed behavior. In the case of the RR model, a correct replica is one that has received the most recent write and has not been rolled back.*

Property 2 (Durability)**.** *Durability is guaranteed if any operation that updates the state of the system survives any combination of faults that is allowed by the RR model. In our case, this means that even if M_R replicas are rolled back, there will be at least one replica with the up-to-date value.*

Property 3 (Operational Liveness)**.** *Generally, a system is live if all operations it supports eventually conclude. We consider a more granular property of operational liveness, which separates the liveness with respect to two classes of operations that are normally defined in replicated systems: read-only (or simply read) operations, which query but do not modify the replicated state, and update (or write) operations that may operate on that state to create a new system state or simply overwrite it.*

Using these properties, we place constraints on the composition of the quorum systems. We differentiate read quorums, of size R_Q, which are sufficient to conduct read operations, from write quorums, of size W_Q, for write operations.

Freshness. We start by observing that, for a given read operation and within the entire replica set, the number of nodes that could possibly have their state rolled back is $\min(s, M_R)$. This means that a read operation has access to a pool of replicas where $W_Q - \min(s, M_R)$ are up-to-date and $N - W_Q + \min(s, M_R)$ may be stale. Given that the most recent write operation contacted W_Q nodes, thus bringing them up-to-date, we derive the following minimum size for a read quorum:

$$R_Q > N - W_Q + \min(s, M_R) \tag{3.1}$$

In our derivation, we turn the inequalities into equalities by adding a positive (or in some cases non-negative) Δ parameter, which captures by how much each variable is larger than strictly necessary, in this case:

$$R_Q = N - W_Q + \min(s, M_R) + \Delta_R \qquad \Delta_R > 0 \tag{3.2}$$

Durability. Additionally, we note that, since M_R replicas can be rolled back, surviving such a rollback implies that a write quorum must include more than M_R replicas, i.e.,

$$\begin{aligned} &W_Q > M_R \\ &W_Q = M_R + \Delta_W \qquad\qquad \Delta_W > 0 \end{aligned} \tag{3.3}$$

Write Liveness. We must also guarantee liveness for write operations. This requires that a write quorum is available despite F unreachable nodes. This is guaranteed provided that:

$$\begin{aligned} N - F &\geq W_Q \\ N &= W_Q + F + \Delta_N \qquad\qquad \Delta_N \geq 0 \end{aligned} \tag{3.4}$$

Final Derivation. The formulae above allow us to arrive at a precise formulation for the system and quorum sizes. In particular, by combining 3.3 and 3.4, we obtain:

$$N = M_R + F + \Delta_W + \Delta_N \qquad\qquad \Delta_W > 0, \Delta_N \geq 0 \tag{3.5}$$

Then, by replacing W_Q (3.3) and N (3.5) in Equation 3.1, we obtain the following equation for read quorums.

$$R_Q = F + \min(s, M_R) + \Delta_N + \Delta_R \qquad\qquad \Delta_R > 0, \Delta_N \geq 0 \tag{3.6}$$

Read Liveness. We conduct the analysis of the liveness conditions for read quorums separately, since these are dynamic conditions, namely due to their dependency on the current number of possibly stale nodes, s. As such, we introduce another dynamic value: f', the number of replicas that are unreachable at any given point. This allows us to express the dynamic liveness condition for reads as follows:

$$N - f' \geq R_Q \Leftrightarrow f' + \min(s, M_R) \leq M_R + \Delta_W - \Delta_R \tag{3.7}$$

This equation allows us to reason about the liveness for reads, depending on specific runtime conditions and on how the static parameters are chosen. For instance, we could require reads to be live in the worst possible case of $f' = F$ and $s = M_R$, yielding $\Delta_W \geq \Delta_R + F$.

However, this is a conservative assumption. An example of a more realistic one would be to assume that we forfeit read liveness in the event of a worst case level of unreachability ($f' = F$) and there is at least one rollback ($M_R \geq 1$). This yields $\Delta_W \geq \Delta_R + F - r$. We also choose to set $\Delta_N = 0, \Delta_R = 1$ to minimize replication costs, allowing us to derive the value for Δ_W from Equation 3.7:

$$\Delta_W \geq \Delta_R + F - M_R \tag{3.8}$$

$$\Delta_W \geq 1 + F - M_R \tag{3.9}$$

When also taking into account that $\Delta_W > 0$, and turning the inequality into an equality to minimize replication costs, this allows us to derive:

$$\Delta_W = \max(1, 1 + F - M_R) \tag{3.10}$$

$$\Delta_W = 1 + \max(F - M_R, 0) \tag{3.11}$$

This results in these possible deployment parameters:

$$N = \max(M_R, F) + F + 1 \tag{3.12}$$

$$W_Q = \max(M_R, F) + 1 \tag{3.13}$$

$$R_Q = F + \min(s, M_R) + 1 \tag{3.14}$$

Atomic update operations. As we mentioned, more complex systems such as replicated databases, instead of following a simple read/write interface, support rich operations that read the most recent value of the system and update it with a new value derived from the value that was read. To achieve this, their protocols may need to gather both a read and a write quorum (i.e., $\max(R_Q, W_Q)$ replies). We dub these quorums super quorums and use S_Q to represent their size.

Quorum properties. This derivation of the various quorum sizes leads to the following set of properties that RR quorum systems obey (also illustrated in Figure 3.2):

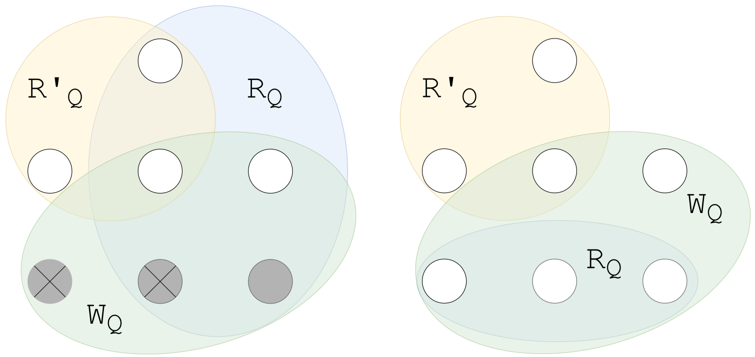
I1. Any read quorum intersects with any write quorum in at least one replica whose state was not rolled back;

I2. It is possible some pairs of read quorums do not intersect.

Using these properties, we can derive the following property of super quorums:

I3. The intersection between a super quorum and a quorum of any other type is non-empty and includes a replica whose state has not been rolled back.

These properties, along with the fact that read quorums can be smaller than write quorums in the normal case when there are no restarts, play a role in the design and performance of protocols in the RR model, as we will see next.



(a) (b)

Figure 3.2: Example of a partially intersecting RR quorum system ($M_R = 4$, $F = 2$). In (a), there were 3 restarts (shaded) and 2 rollbacks (crossed). Thus, the read quorum receives suspicious replies and grows larger, ensuring that both read quorums still intersect W_Q in at least one up-to-date replica (as required by **I1**). In (b), there are no restarts (and, by extension, no suspicion of rolled back state), and thus read quorums can be disjoint, as noted by **I2**.

Classifying RR Quorum Systems Reviewing the quorum properties in the previous paragraph, there is one property that stands out: **I2**. In quorum literature, the intersection of quorums is a key property, but in RR, it naturally arises that, for some parameterizations, this is not guaranteed to happen between all read quorums. Here, two disjoint read quorums may in general reflect different system states — a phenomenon usually described as split brain. This can happen when an operation that changes the system state is still in progress (or even halted, due to the crash of the initiator). Handling this peculiarity will require some protocol support, creating mechanisms to ensure that a single value is reported by all read quorums. However, in certain parameterizations, property **I2** can be guaranteed never to hold, and thus such extraneous protocol infrastructure can be avoided. We classify RR quorum systems in two categories accordingly:

Definition 1 (Partially Intersecting RR Quorum System)**.** *A RR Quorum System in which there are two read quorums, R^1_Q, R^2_Q such that $R^1_Q \cap R^2_Q = \emptyset$. In other words, where **I2** holds.*

Definition 2 (Fully Intersecting RR Quorum System)**.** *A RR Quorum System in which for all pairs of read quorums, R^1_Q, R^2_Q $R^1_Q \cap R^2_Q \neq \emptyset$ is true. In other words, where **I2** does not hold.*

Theorem 1 describes how to characterize a RR quorum system as either partially or fully intersecting based on its parameterization:

Theorem 1 (Characterization of RR Quorum Systems)**.** *A RR quorum system with parameters $M_R, F, \Delta_R, \Delta_W$ and Δ_N, where $\Delta_R > 0, \Delta_W > 0$ and $\Delta_N \geq 0$ and in which the total number of replicas is $N = M_R + F + \Delta_W + \Delta_N$, the size of read quorums is $R_Q = F + \min(s, M_R) + \Delta_N + \Delta_R$ (with s as defined* *above) and the size of write quorums $W_Q = M_R + \Delta_W$ is fully intersecting if and only if:*

$$F + \Delta_N + 2\Delta_R > M_R + \Delta_W \tag{3.15}$$

Proof 1. *To prove the intersection of any two read quorums, it suffices to show that Equation 3.15 is equivalent to $2R_Q > N$, even when $s = 0$. This in turn is equivalent to stating that the dimension of* *any* *two read quorums from the set of the smallest read quorums possible for a quorum system (i.e., those where $s = 0$) is larger than the size of the quorum system. This implies that at least one replica is present in both read quorums, and as such property* ***I2*** *cannot hold. We denote the size of the smallest read quorums as $R_Q(s = 0)$.*

$$\begin{aligned} 2R_Q(s=0) &> N \\ \Leftrightarrow 2F + 2\min(0, M_R) + 2\Delta_N + 2\Delta_R &> M_R + F + \Delta_W + \Delta_N \\ \Leftrightarrow F + \Delta_N + 2\Delta_R &> M_R + \Delta_W \end{aligned}$$

□

Corollary 1. *In the parameterization in Equation 3.12, the associated quorum system is fully intersecting if and only if $F \geq M_R$*

Proof 2. *The parameterization in question fixes $\Delta_R = 1, \Delta_W = 1$ and $\Delta_N = 0$. Substituting in Equation 3.15, we get:*

$$\begin{aligned} F + 2 &> M_R + 1 \\ \Leftrightarrow F + 1 &> M_R \\ \Leftrightarrow F &\geq M_R \end{aligned}$$

□

Reflections on the parameterization of the system. RR quorums are more complex than regular quorum systems. They include two parameters (M_R and F) instead of one, with an additional runtime value (s). Moreover, the quorum system is inherently asymmetric, leading to three different types of quorums (read, write, and superquorums), as opposed to a single one. This puts a burden on the system designer to use the right type of quorums for different protocol steps, and also, at deployment time, to consider how to choose M_R (to represent the anticipated maximum number of simultaneous rollback attacks on the system), and F (to encode the availability of the system, by estimating how many replicas can crash without jeopardizing liveness). However, we believe that this complexity is warranted, not only because it is naturally derived from the nature of the faults (in particular, the fact that it is possible to determine precisely when a replica is or is not susceptible to a rollback of its persistent state), but also because it allows us to extract the maximum performance from the system and avoid wasting resources used in replication.

Comparison with existing asymmetric quorum schemes. Previous proposals for asymmetric quorum schemes differ significantly from RR quorum systems. The idea of asymmetric quorums dates back to one of the initial proposals for quorum systems by Gifford [75], where a replicated object has a certain number of votes and in order to read the value, r votes must be gathered, whilst w votes are required to write the value. The equivalent to the intersection property **I1** is guaranteed by ensuring that $r + w$ exceeds the total number of votes of the object. This use of asymmetric quorums has also been motivated by different goals. In particular, asymmetric quorums have been proposed in the context of asymmetric trust assumptions [76], where each node makes its own assumptions about which nodes might be Byzantine. Another type of use of asymmetric quorums is to obtain better performance, by making the commonly used quorums smaller than the ones that are used less often [26, 27, 77] or reducing the asymptotic complexity of quorums at the expense of more replicas [78].

Compared to these approaches, RR quorums are derived from the dynamic nature of the number of possible rollbacks that are present in the system at any given moment. This natural construction leads to interesting properties of the system, namely allowing for performance to improve in the normal case when there are no recent replica restarts, due to the use of relatively small read quorums.

3.3 Replication protocols

Designing and implementing new replication protocols, as well as proving their correctness, is a non-trivial effort. Consequently, rather than building protocols for the RR model from scratch, we propose a set of principles for adapting existing protocols to the fault model. In this section, we identify principles for this adaptation and apply them to existing protocols, showing that the adaptation is straightforward.

3.3.1 Principles and challenges for protocol adaptation

When adapting existing protocols to the RR model, it is convenient to start from a crash fault tolerant protocol. Due to the correctness of execution of nodes in the RR model, nodes observe mostly crash fault behavior; the only additional fault behavior is that their externally stored state may be stale after a restart.

Most replication protocols are quorum-based, where a client (or a replica acting on behalf of a client) needs to obtain responses from the quorum or replicas to perform an operation. Therefore, an important aspect of adapting a crash fault tolerant protocol consists of following the quorum sizes defined in Section 3.2.3. Similarly, reconfiguration is a key aspect of practical replication protocols. Quorum sizes for these reconfiguration protocols also need to be adapted accordingly, either to be a write quorum or a super quorum, depending on whether the update needs to be atomic.

Each node must be augmented to maintain a suspicion flag. When a node restarts, it set the suspicion flag to true. At this point, the node runs a recovery subprotocol (eagerly or lazily), which is not required in the crash model.

While the recovery subprotocol is dependent on the specifics of the protocol being modified, the general method is as follows. On restart, a replica queries a read quorum of replicas for a digest of their state, retaining the replies that contain the most recent state, e.g., determined through timestamps as exemplified next. From this read quorum of digests, it can determine the digest of the current system state, and check whether its state is up-to-date, thus clearing the suspicion flag. If not, then the specific parts of the state that are stale need to be fetched from other replicas to bring the recovering replica up-to-date. To efficiently find which elements of the state are out of date, replicas may maintain a Merkle tree, which allows for determining which parts of the state need to be fetched without transmitting a large amount of information. Note that this subprotocol can run lazily and in the background, while the replica continues satisfying requests. Doing so might allow for clearing the suspicion flag in case of an update where the new state does not depend on the previous version.

If the quorum system is fully intersecting, these changes are enough. Otherwise, mechanisms that ensure the correctness of reads from possibly disjoint read quorums need to be put in place. These mechanisms will be generally protocol specific.

Next, we will present two example protocols that follow these principles and illustrate some of the above challenges.

3.3.2 Read-write register

In this section, we present an adapted version of the ABD [1] protocol for a distributed read/write register with linearizable semantics[1] [79] under the RR model. ABD provides a simple read/write interface, which is useful for storage systems or services that offer a read/write interface. Read/write register protocols have the advantage of guaranteeing termination even in asynchronous systems with faults, and

---

[1] A concurrent object is linearizable iff there exists a serialization of all operations which is equivalent to a sequential execution and the serialization matches the real-time order of invocation/reply.

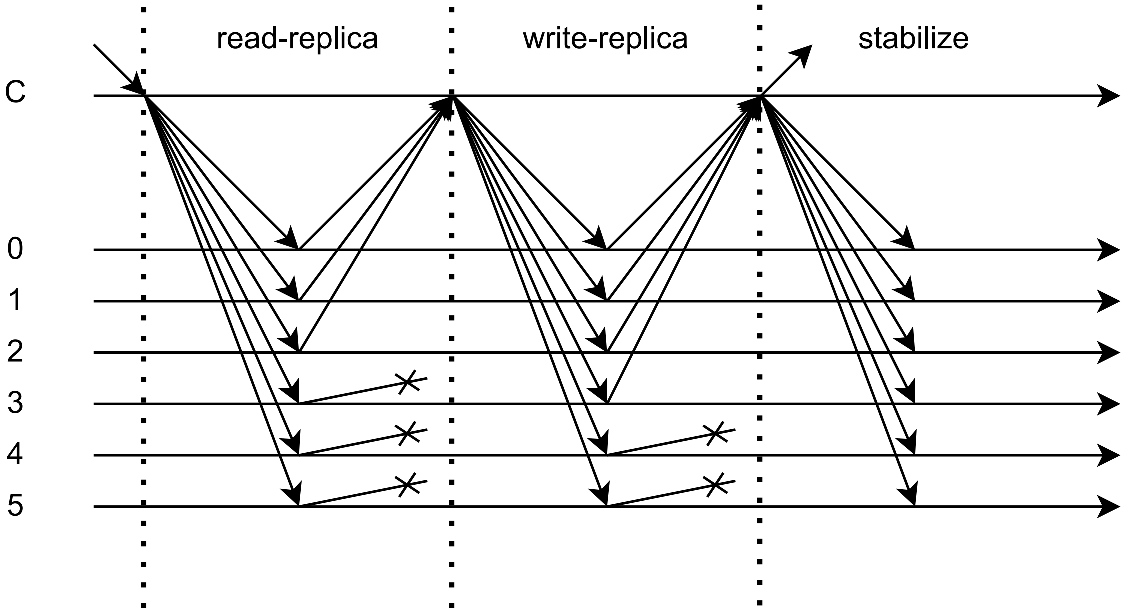


Figure 3.3: Time diagram of the register write operation.

having good performance due to a simple message pattern that is linear in the number of replicas [80]. Figure 3.5 shows the pseudocode for the read operation, while the write logic is shown in Figure 3.6. Since the code follows closely the original ABD protocol, our explanation highlights where we adapted the protocol to the new fault model. We also provide time diagrams of the operations in Figures 3.4 and 3.3.

Timestamp Structure. Each data item stored is associated with a timestamp, which defines the linearization order of the version that is stored. Timestamps have two numeric components $\langle seqno, client\_id\rangle$, where $client\_id$ is the unique, ordered id of the client issuing the write. This breaks ties when two clients write different values to the same sequence number.

Write. This operation is similar to the ABD protocol, but uses the RR quorum system. In the first round a read quorum is gathered to discover the most recent sequence number (the one associated with the highest timestamp). In the second round, that sequence number is incremented by the client, appended with the client id, and the resulting timestamp is sent with the value to be written. Upon receiving this second round message, replicas overwrite values if the received timestamp is greater than the one associated with the data they store. This clears the suspicion that may have existed of the value of the register, as the written value is guaranteed to be fresh.

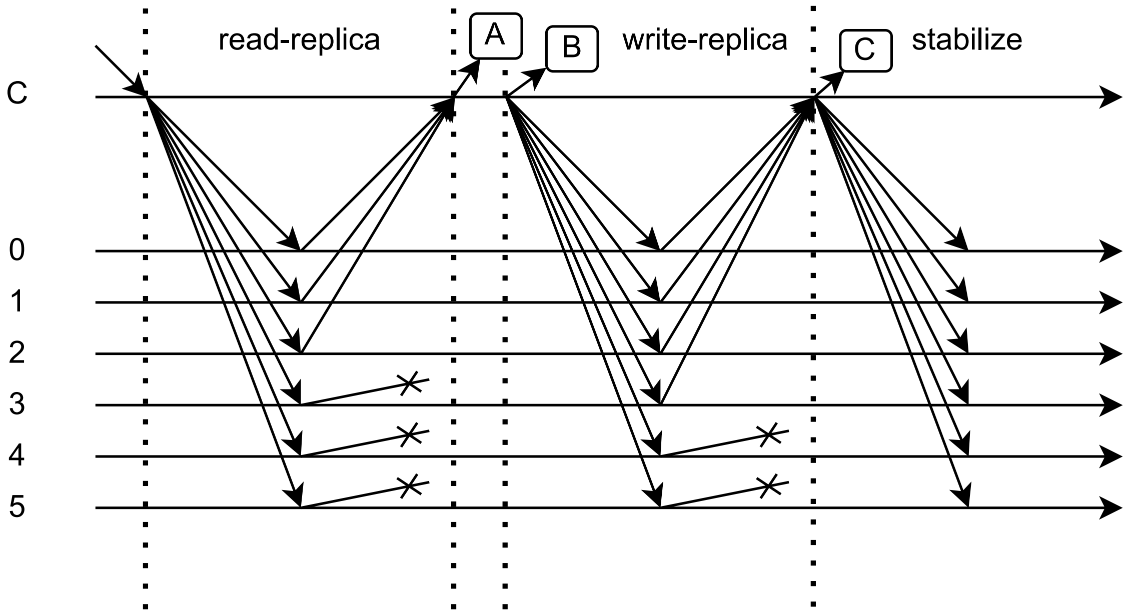


Figure 3.4: Time diagram of the register read operation. If the value is stable, the read can return at point A. If the the write quorum gathered in the first phase is unanimous, the read can return at point B. In this case, for clarity, the stabilization is omitted in the figure. Else, the most recent value is written back to a write quorum, and then the read operation returns at point C, issuing an asynchronous stabilization phase.

Read. Following the ABD protocol, reads occur in one round in the common case, with a second round being required if a value needs to be written back. In particular, in the original ABD protocol, the first round queries replicas for their data and timestamp, waits for replies from a majority (which equates to both a read and a write quorum) and the return value corresponds to the reply with the highest timestamp. However, when there is no majority that holds that timestamp value, the second phase is required, writing this timestamp and data to a majority. (This is needed to conclude a write operation that executes concurrently or was left unfinished.)

Translating the notion of a majority to read and write quorums in the RR model presents a subtle challenge in the context of partially intersecting quorum systems. Even though intuitively the initial read round only requires a read quorum (and in fact this is sufficient to determine the read reply), the optimization of skipping the second round is only applicable if there is unanimity for that timestamp in a write quorum. This is because read quorums do not necessarily intersect (property **I2**), which implies that contacting only a read quorum would allow for a "split brain" situation, where clients read different values depending on which quorum they contact, thus breaking linearizability. The problem with waiting instead for a super quorum, however, is that in scenarios where inter-node latency has a wide variance such as geo-replication, this introduces an additional latency that erodes the performance advantage of the smaller quorums.

```
Read at client $c$
    send READ-REPLICA to all replicas
    wait until received either $R_Q$ replies, such that, for the highest timestamp $ht$, $ht.stable ==$ TRUE or until
        received $W_Q$ replies where the highest timestamp has $ht.stable ==$ FALSE
    if $ht.stable ==$ true then
        return SUCCESS$(ht, val(ht))$
    if $\exists W_Q$ of replies with $ht$ then
        send STABILIZE$(ht)$ to all replicas;
        return SUCCESS$(ht, val(ht))$
    send WRITE-REPLICA$(ht, val(ht))$ to all replicas
    wait until received $W_Q$ of SUCCESS replies
    send STABILIZE$(ht)$ to all replicas
    return SUCCESS$(ht, val(ht))$
```

Figure 3.5: Pseudo-code for the register read operation.

```
Write (value $v$) at client $c$
    send READ-REPLICA to all replicas
    wait until received $R_Q$ replies
    let $ht$ be the largest timestamp in quorum
    let $new\_ts$ be $\langle ht.seqno + 1, c, \texttt{false} \rangle$
    send WRITE-REPLICA$(new\_ts, v)$ to all replicas
    wait until received $W_Q$ SUCCESS replies
    send STABILIZE$(ht)$ to all replicas
    return SUCCESS
```

Figure 3.6: Pseudo-code for the register write operation.

Stabilization. To address this issue, we introduce an extra asynchronous phase, called the stabilization phase. This phase takes place in the background after a value has been successfully written to a write quorum (either in a write operation or in the writeback phase of a read operation), without blocking the operation from returning to the client. A STABILIZE message is sent to the replicas so that they will set a stable flag associated with the recently written timestamp and value. Since this is an optimization, there is no need for replicas to reply to this message. Marking a value as stable means that the write operation for this value has concluded (i.e., reached a write quorum), which implies that all read quorums include at least one replica that will report either this or a newer value, given that a write quorum intersects all read quorums (**I1**). Therefore, in the first phase of the read operation, if the most recent value in a read quorum is marked stable, even if it is read from a single quorum replica, it can be immediately returned, since the stable flag indicates that it has been written to a write quorum and will therefore be seen by any subsequent read quorum, thus obeying linearizable semantics.

The stabilization phase can occur at any point after the write concludes. Considering an eager approach, the stabilization would occur right after the value has been written. We follow a lazier option, triggering the stabilization only after the first read (thus avoiding stabilizing a value that is never read).

This mechanism can be altogether skipped if the quorum system is fully intersecting, as the intersection of read quorums guarantees that the value present in any read quorum is correct.

```
Execute (operation $op$)
  1. client_send EXECUTE($op$) to leader
  2. leader assigns slot number $s$ to $op$
  3. leader_broadcast PREPARE($s, op$), after logging to disk
  4. replica_broadcast ACCEPT($s, op$), after logging to disk
  5. wait until $\#accepts(s) \geq S_Q$, marks $op$ as accepted
```

Figure 3.7: Pseudo-code for the normal case SMR operation.

Proof sketch. We prove the correctness of the resulting protocol in the appendix, and sketch here a correctness argument. The proof of linearizability of a read/write object follows a helper theorem [81], requiring, for any execution, the existence of a total order that is both consistent with the results the operations return and consistent with the real time order of request invocations and replies. In our proof, this total order is established by the timestamp order (in case of operations with the same timestamp, reads follow both writes and other reads that precede them in real time order). Then we prove that this meets both consistency requirements above: for the output of reads, this is by construction due to reads being ordered after the corresponding writes; then, the consistency with the real time request order follows from the fact that the quorum intersection property **I1** from Section 3.2 implies that reads see the effects of previously completed reads or writes either directly, because the preceding operation contacted a write quorum, or indirectly, via the stable flag.

3.3.3 State-machine replication

State machine replication (SMR) allows for replicating any deterministic service by enforcing that operations are executed in the same order on all replicas. Paxos [2] is the best known instance of this paradigm, but there are several different descriptions, with little consensus on what the exact Paxos protocol entails. Since our goal is to showcase the changes required by the RR model, we chose as a starting point the versions that describe the persistent state that is logged at each protocol step [34, 82]. In contrast, most other Paxos descriptions only store the replica state in memory, and therefore do not allow, for instance, the simultaneous restart of all the replicas — only up to F of them can restart at a time. We next present the adaptation of this protocol to RR, focusing on the normal case operation for conciseness.

Normal case operation. We follow the multi-Paxos protocol [82, 83], where there is a component of the protocol that elects a stable leader replica, guaranteeing that only that replica can propose the sequencing of new operations while it remains the leader. That sequencing corresponds to the normal case operation, which is described in the pseudocode in Figure 3.7, and is depicted in Figure 3.8 (following the protocol description from [34]). In summary, the leader replica proposes a sequence number

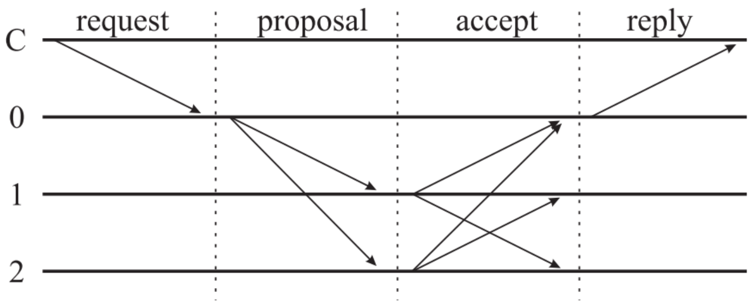


Figure 3.8: Diagram of a normal case SMR operation

for incoming client requests, and this sequencing must be accepted by a quorum of replicas (a majority in Paxos), who validate that this sequence number had not been assigned yet. Once such a majority is gathered, replicas execute the client operation in that order and the leader replicas replies with the output of the operation to the client.

Our protocol follows directly from this protocol description [34], modifying only the quorum size when gathering the quorum of accept messages. When moving from majoriy quorums to the separate quorum types, we need to observe that a Paxos round is both reading and writing the current state of the Paxos protocol. This is because it must read that the slot that is being proposed is not yet taken, and at the same time record the fact that the slot becomes taken and cannot be used in subsequent proposals. Thus, majorities are replaced with superquorums in the RR-tolerant version of Paxos.

Leveraging the RR model. So far, the protocol adaptation does not leverage the small read quorums enabled by the RR model. Even read-only operations are serialized in the state machine, and as such need to update the system state, namely to fill the position in the sequence of operations.

To leverage small read quorums, we adapt the read-only optimization described in some protocols such as PBFT [21] or the Paxos description by van Renesse and Altinbuken [84]. In this optimization, the client contacts a read quorum with an OPTIMIZED-READ message, asking for the result of executing the read-only command against current state of the replicas. If the replies are unanimous, the client can return the value. Figure 3.9 showcases the workflow of a read-only operation.

Applying this optimization requires careful reasoning to avoid violating the linearizable semantics of the protocol, particularly in the context of partially intersecting quorum systems. To understand why, consider the possibility of two successive read operations, r_1, r_2, where r_1 ends before r_2 begins, and that use quorums Q_{R1} and Q_{R2} (respectively) and execute concurrently with a write w that gathers a quorum of accepts Q_W. In this case, and given property **I2** (read quorums may not intersect), r_1 may see w as being complete in a read quorum, but r_2 only contains replicas that have not yet gathered a write quorum of accepts for w, since those replicas might have sent but not yet received the necessary number of accepts. This would violate linearizability since r_1 precedes r_2 in real time but, given their

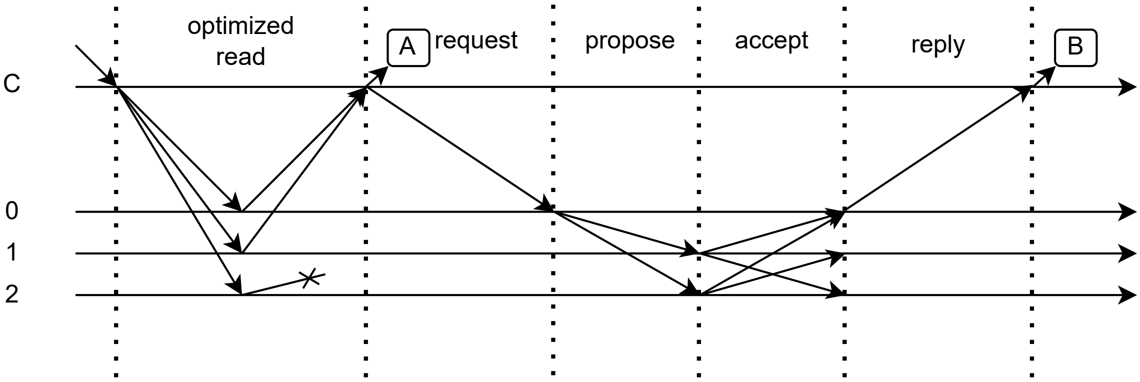


Figure 3.9: Time diagram of the SMR read-opnly operation. If the optimized read is successful, the operation returns at point A. Otherwise, it falls back to a regular SMR operation, returning at point B.

outputs, they cannot be linearized in that order.

This is yet another occurrence of a split brain scenario, but the solution in this situation is different: instead of confirming a written value via stabilization, we abort the optimized read when it is possible that another value has been written to the state machine, falling back to reading using a state machine operation. A replica can detect this situation if it has sent an ACCEPT message for a slot higher than the last executed operation (as it indicates the possibility of another read quorum with a different value). If no replicas in a read quorum have done this, then no other operation has concluded (**I1**). As was the case in the register protocol adaptation, this mechanism can be skipped in the case of fully intersecting quorum systems.

Proof sketch. Just as in the ABD protocol, we sketch the proof based on the existence of a total order for the operations that is consistent with both the output of operations and the real time order of request execution and replies [81]. This total order is built in the same manner as before, i.e., it is given by the slot number s, breaking ties by having read-only operations succeeding both the most recent read/write operation reflected in the reply and read-only operations with the same slot number that precede them in real time. By the construction of the protocol, this order is consistent with the results that are output to the client, since the replies reflect the execution of the preceding sequence of state machine commands. The proof for that the order is consistent with the real time order of requests is straightforward for the non-optimized protocol but more subtle for the case where one or both of the requests follow the read-only optimization. In this case, a later read-only operation cannot revert to a previous state because of the protocol feature that replicas with pending accepts deny an optimized read. This ensures that it is impossible to have a more recent state that could have been reflected by a preceding read/write or optimized read, because intersection property **I1** implies that at least one node from the read quorum in the later read would have sent the accept for the operation that created the more recent state, since its execution requires a write quorum of accepts.

3.3.4 Security Properties of TEE replication

If the threat model for TEEs explained in Section 3.1.1 holds, the protocols described in this section achieve freshness, integrity and confidentiality. Confidentiality of the overall system is inherited trivially from the TEE fault model. Integrity and freshness of data follow from the correctness of the protocols, guaranteed by their linearizability proofs (present in full in the Appendix). These proofs rely on the intersection properties of the quorum system, in particular that they mask the rolled back replicas. Moreover, they assume the RR model applies to the replicas, which is guaranteed by encapsulating replicas inside the TEEs. Crucially, the usage of TEEs (which have integrity of the computation) guarantees that the protocol is followed by all replicas (even if they have stale data).

3.4 Evaluation of TEE replication protocols

We evaluate our various implementations based on the RR model using micro-benchmarks for the protocol implementations. Our experiments attempt to answer the following questions:

1. How significant is the effort to change a CFT implementation to support the RR model? (§3.4.3)
2. How do the protocols based on the RR model compare with their counterparts based on the Crash and Byzantine fault models? And how does that performance behave under increased load? (§3.4.4)

3.4.1 Implementation

We implemented both the read/write register and SMR protocols in the three fault models (crash, RR, and Byzantine) in Intel SGX (version 1), using C++. All prototypes were implemented using the same codebase, limiting the changes between prototypes to what was required by the protocols (e.g., extra protocol steps, different quorums). The PBFT [21] implementation uses the standard optimization of using MACs instead of digital signatures.

SGX applications have two separate regions of memory: the application (untrusted) and the enclave (trusted). In all cases, the application code comprises 1KLoC (mostly for bootstrap and interfacing with the local OS). All replicas are implemented using a single-threaded event loop, and take between 4.5KLoC for the distributed register and 4KLoC for SMR. Additionally, the client libraries, which interact with the replicas take up 5KLoC (distributed register) and 4KLoC (SMR).

All prototypes are open-sourced under the MIT license and available at `https://gitlab.mpi-sws.org/restart-rollback`.

3.4.2 Experimental Setup.

We ran our experiments using 7 machines with Intel® Xeon® E-2174G processors running Debian Linux version 4.19 to run the replicas, plus a machines equipped with an Intel® Xeon® Platinum 8260M processor running Debian Linux version 5.4 to execute the clients. All machines were connected to same local network. To emulate different deployment scenarios, we developed `sloth` [85], which internally uses `netem` [86] to implement a network topology from a high-level JSON description. We considered two deployment scenarios in these experiments: a local area network where all machines are connected by links whose latency follows a normal distribution with mean 1ms and standard deviation of $\sigma = 0.5$ms (which we will refer to as the 1ms topology); the deployment scenarios of Figure 4.3; and a geo-replication scenario, based on the measured the link properties (latency and bandwidth) between several AWS EC2 [87] instances (t2.medium or t3.medium types), located in regions spread across the globe (AWS topology). This setup ensures flexibility and experimental reproducibility.

In our result graphs, each data point represents the median measurement over $3,000$ requests (except throughput numbers as described belows) and the error bars show the 5^{th} and 95^{th} percentiles.

3.4.3 Code changes

Changing a CFT implementation to reflect the protocol changes required by the RR model requires low programming overhead. Mainly, this consists of handling the restart flag, the existence of different quorums and the mechanisms to prevent split brain (e.g., the stabilization in the register protocol). For the read/write register prototype, to implement the RR protocol we modified 72 LoC and added 211 new ones to the CFT implementation. For the RR SMR prototype, we modified 24 LoC and added 28 to the CFT prototype.

3.4.4 Protocol performance in different models

Next, we focus on comparing the performance of both classes of protocols under different fault models. We compare our adapted read-write register with the original ABD protocol and the BFT read-write register protocol described in [72]. For SMR, we compared our protocol with Paxos as described by Kirsch and Amir [34] and the PBFT protocol [21]. We fixed the fault parameter, $F = 2$ across quorum systems (with $M_R = 2$ in the RR quorum system), yielding quorums with 3 replicas in the crash and RR models and 5 replicas in the Byzantine model. The parameterizations are summarized in Table 3.1, along with the resulting quorum and system sizes.

The results in Figures 3.13- 3.16 reflect the differences between the different fault models. We observe that larger quorums make the operations slower, since the operation latency is bound by the latency of the slowest replica in the quorum. Notably, in both cases the performance of the RR protocols

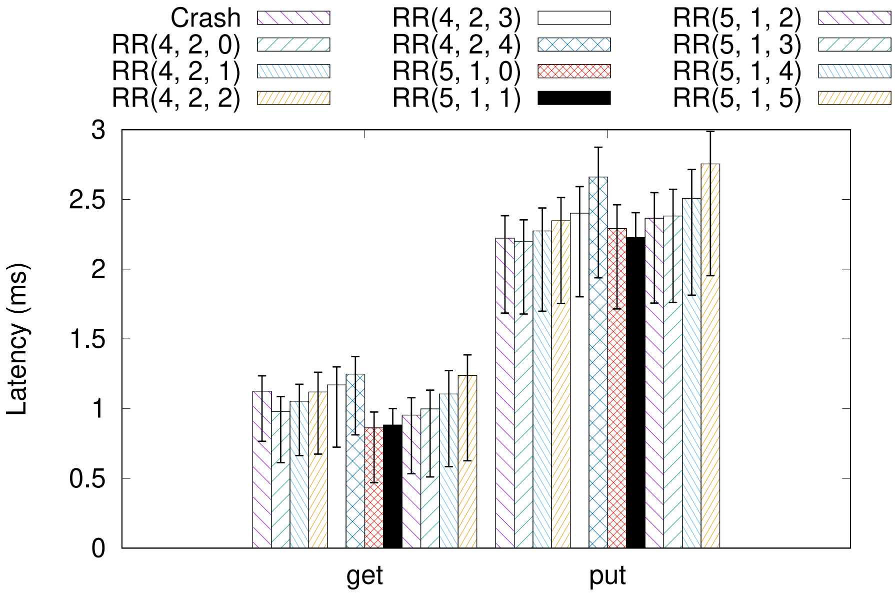


Figure 3.10: Read-write register

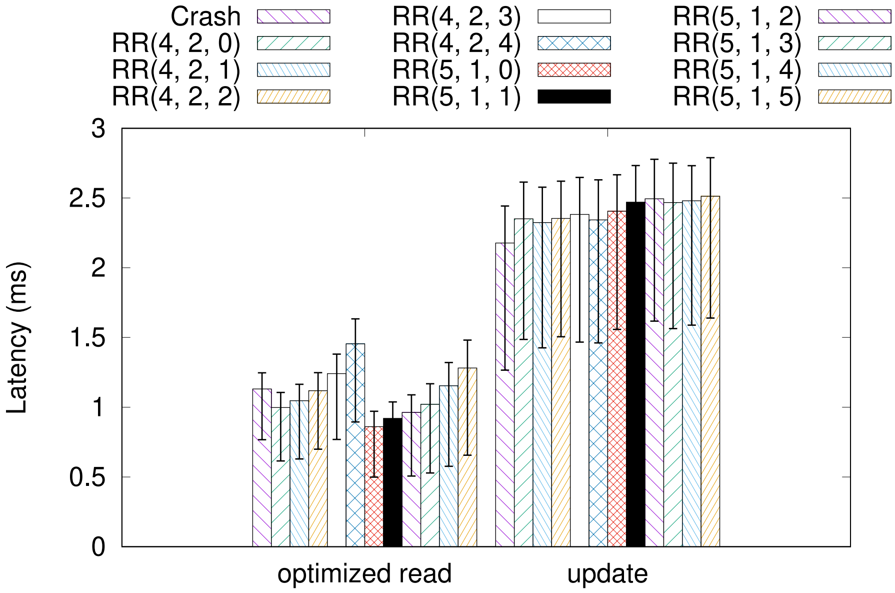


Figure 3.11: State machine

Figure 3.12: Latency for different configurations and models in the 1ms topology. CFT uses $F = 3$ and $RR(M_R, F, s)$ denote RR with parameters M_R and F, with s restarts.

matches that of the CFT ones, which is to be expected as they have equally sized quorums.

Next, we measure how the performance of different quorum systems degrades as the system load increases, as well as the maximum throughput obtained. In the experiment, we vary the offered load by increasing the number of concurrent client requests of a single type, and measure both latency and throughput, keeping the message and object sizes fixed. Throughput is measured by counting the

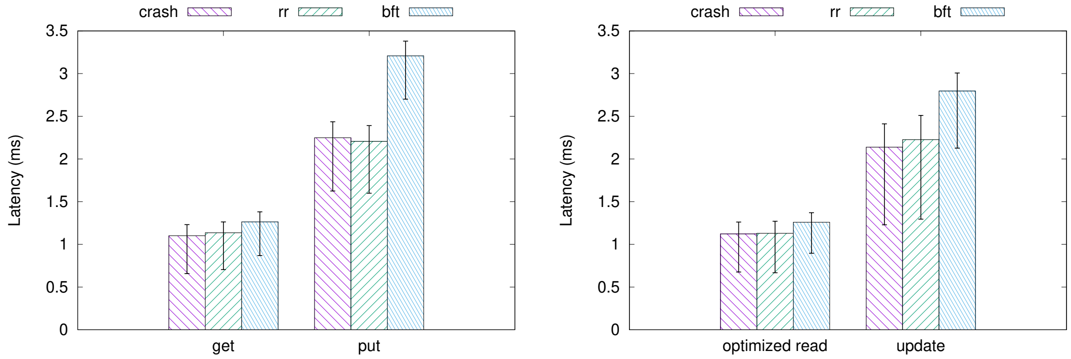


Figure 3.13: Read-write register ($1ms$)

Figure 3.14: State machine ($1ms$)

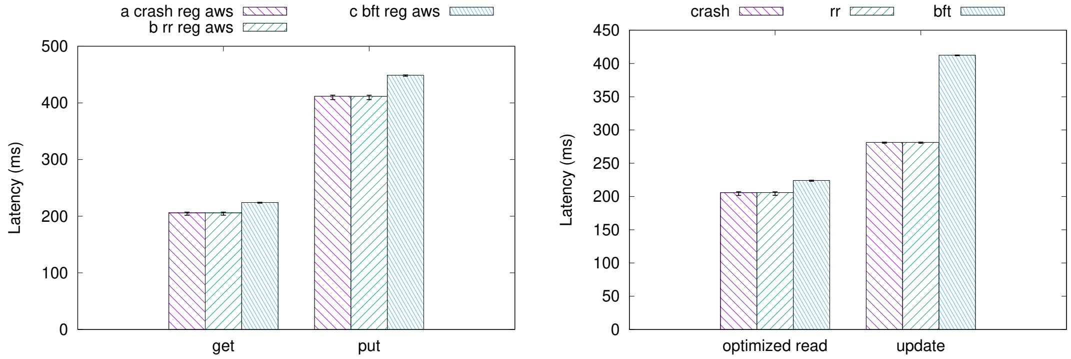


Figure 3.15: Read-write register (AWS)

Figure 3.16: State machine (AWS)

Figure 3.17: Operation latency for different protocols in different network topologies

number of replies obtained per time interval. Each data point corresponds to the median latency or throughput over 5 seconds of continuous load after a warm-up.

Figure 3.18 shows that the read-write register with the RR configuration achieves a maximum throughput of approximately 200 and 110 Kops/s for reads and writes, respectively, being matched by the CFT register, as expected. The BFT register has significantly lower throughput, peaking at approximately 30 and 2 Kops/s for reads and writes.

In Figure 3.19 we can again observe that the CFT and RR protocols behave comparably, peaking at approximately 65 and 200 Kops/s for updates and optimized reads, respectively. The BFT protocol peaks at 20 and 160 Kops/s for updates and optimized reads, respectively. This is due to both larger quorums and the extra protocol round of PBFT.

Model	N	F	M_R	$R_Q(s = 0)$	W_Q
CFT	5	2		3	3
RR	5	2	2	3	3
BFT	7	2		5	5

Table 3.1: Parameters required by different fault models.

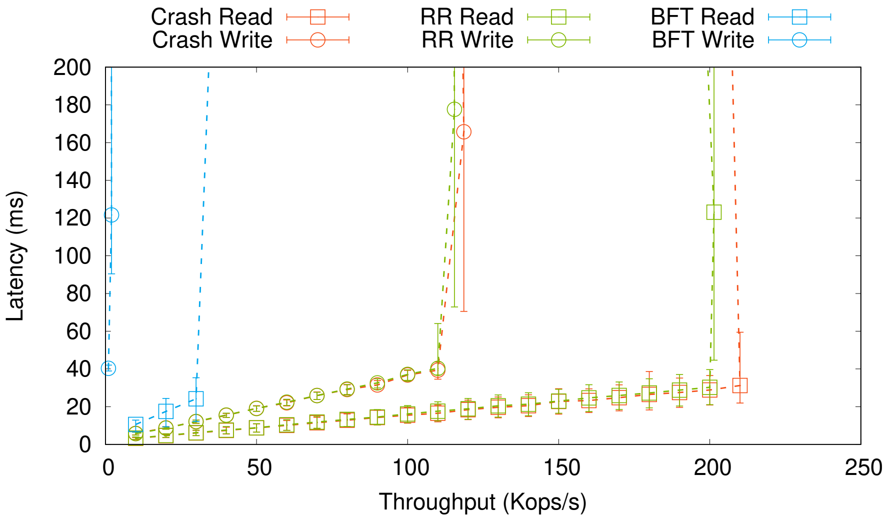


Figure 3.18: Read-write register

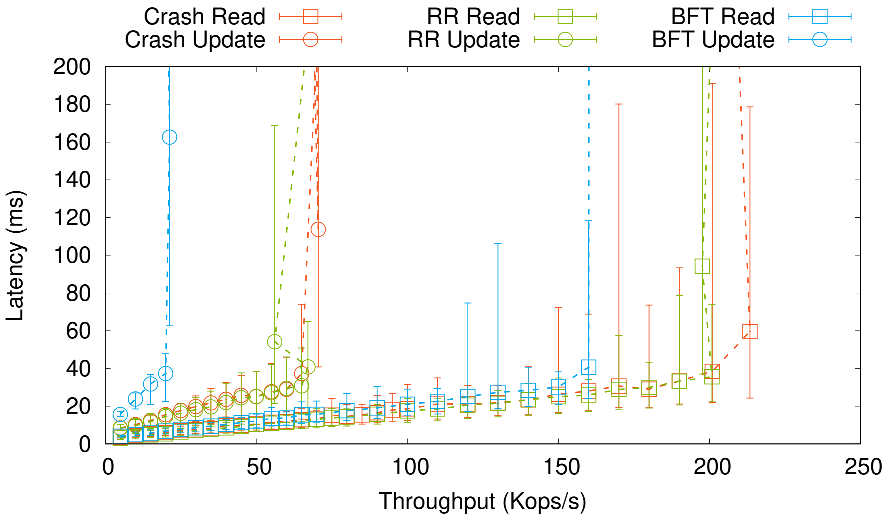


Figure 3.19: Replicated state machine

Figure 3.20: Throughput-Latency curve for different protocols

In all cases, the throughput becomes bounded by the resources of the nodes: the amount of request that nodes need to process exceeds the capability of the hardware. In the BFT protocols, there are both more replicas and more rounds, which equates to more messages to be processed and, by extension, lower throughput.

In the preceding experiments, the crash and RR protocols used equally sized quorums, and as such had very similar performance. However, as discussed in Sections 3.2 and 3.3, the RR model has asymmetric quorums, which lends itself to faster reads (at the expense of slower writes). Moreover, in the preceding experiments the number of restarts has been set to 0, as this is the common case. To better explore the configuration space of RR quorums and their performance difference to CFT, we reran the latency experiments from Figures 3.13–3.14, but with different quorum configurations. In Figures 3.10–3.11, we can observe that, by leveraging the smaller quorums of RR, read operations become faster than their equivalents in CFT, at the expense of more expensive writes. Moreover, as the number of restarts increases, the difference to CFT shrinks, and eventually CFT reads outperform RR, in the uncommon case where most replicas have just restarted and have yet to run their recovery protocol. Similarly, as the number of restarts increases, write/update operations also become more expensive (as they require read quorums in some steps).

Overall, the results show that RR has very close performance to CFT, while offering rollback protection and better read performance in some configurations, in the common case with few restarts. Compared to BFT, RR offers significantly better performance due to its smaller quorums.

3.5 Discussion: Generalization of the RR model

A key insight of the RR fault model is that it is possible to detect, with false positives, when non-crash faulty behaviour has occurred in the system. With this suspicion, plus a deployment specific hard limit on the number of nodes that can actually exhibit this faulty behaviour at the same time, we have derived the RR quorum systems and associated protocols. The RR fault model can be considered a generalization of the crash fault model (i.e., when $M_R = 0$). It is interesting to consider whether it can itself be further generalized to faulty behaviour other than rollbacks.

From a technical point of view, this is definitely possible. Provided that there exists a suspicion function that detects, with false positives, when non-crash faulty behaviour has occurred while reading the system, one could reuse the derivation of quorum systems presented in this chapter and the associated protocols. This would even generalize to the Byzantine model, as a trivial function which always suspects that a fault might have happened would correctly capture Byzantine behaviour. Curiously, the quorum system derived using this function is the one described by UpRight [22]. Another trivial function would never suspect non-crash behaviour, being valid in the crash fault model.

However, outside these trivial suspicion functions and the suspicion function that simply reports the suspicion flag from RR nodes, we have not been able to craft functions that present some usefulness. Although this does not mean, by any means, that they do not exist, our experience points to the suspicion function not being a particularly good abstraction for capturing intermediate fault models between the crash and Byzantine other than the RR model.

4

TEE replication using the RR model

Contents

4.1 TEEMS: Metadata service for trusted cloud storage 45
4.2 Implementation 49
4.3 Evaluation 50

As we have discussed previously, TEEs are an interesting option deploying replicated systems in an untrusted public cloud. In this chapter, we present the TEE Metadata System (TEEMS), a replicated metadata service for trusted cloud storage. TEEMS has a double purpose in the context of this dissertation: it showcases a practical system using the RR fault model and associated register and SMR protocols and provides the means of enhancing existing cloud storage systems such that they can be trusted and remotely attested in the same fault model as TEEs (and, by extension, used by them as a persistent storage, with rollback protection).

This chapter is organized as follows. Section 4.1 provides the motivation and design of TEEMS, followed by Section 4.2, which describes the implementation of TEEMS. Finally, Section 4.3 presents an extensive evaluation of TEEMS using real-world workloads across different deployments.

4.1 TEEMS: Metadata service for trusted cloud storage

4.1.1 Motivation

TEEs have enabled services like Azure Confidential Computing [38] and Google Confidential VMs [39], where Cloud tenants can use compute services without trusting the platform provider. However, the strong security properties from TEEs do not automatically extend to cloud storage. To create a cloud-based system with similar strong security guarantees of TEEs, it is only natural to consider using TEEs as its building blocks. A TEE-encapsulated metadata service, replicated for availability and fault-tolerance, can maintain encryption keys and version information for data blobs stored in untrusted Cloud storage. By ensuring confidentiality, integrity, and freshness of the metadata, the service extends the same guarantees to the encrypted and versioned data blobs. Moreover, by supporting an atomic update operation on metadata, the service enables concurrent sharing of data blobs.

This approach needs to be integrated at the replication level. Fundamentally, even assuming an efficient way for storing the state of a single TEE, this solution would still be unable to support concurrent sharing between multiple TEEs. This is because ensuring freshness of a single TEE's own state relies on persistently storing a monotonic counter [11, 43, 44] or a top-level hash [41], which can only be updated by the TEE itself. Concurrent sharing of persistent state instead requires atomic, concurrent updates of state blobs and their associated metadata (e.g., counter).

We designed and implemented a replicated metadata service called TEEMS (for TEE-based Metadata Service) based on the RR fault model. TEEMS's replicas can be hosted in a diverse set of cloud providers for high availability and resilience. TEEMS provides a read/write interface for metadata and guarantees that readers always receive the metadata (e.g., encryption key) of the most recent version of a data blob. TEEMS also supports access control policies for metadata, and therefore allows clients to selectively share the associated data blobs. The service therefore enables trusted storage services

Return Value		Command and Arguments
{*status* }	←	teems-init(*client ID*)
{*status* }	←	teems-close()
{*status* }	←	teems-write(*id*, val)
{*status*, val, *ver* }	←	teem-read(*id*)
{*status* }	←	teems-delete(*id*)
{*status* }	←	teems-change-policy(*id, policy-code*)

Table 4.1: TEEMS Interface

that extend the strong guarantees of today's TEE-based cloud compute services to persistent storage.

4.1.2 TEE-grade cloud storage with TEEMS

Clients can use TEEMS to lend untrusted cloud storage TEE-level guarantees, by using the API in Table 4.1. TEEMS maintains metadata for each data blob, i.e. a short summary of the most recent version of a data blob, namely its hash and encryption key. The encrypted data blob is then stored in an ordinary cloud storage service. This extends the integrity, confidentiality, freshness and selective concurrent sharing properties of the metadata service to the cloud storage, while relying on the cloud service only for storage and availability.

Concurrent sharing of mutable data fundamentally requires a read-modify-write operation (i.e., some form of SMR). However, state machine operations are more expensive than read/write register operations. TEEMS minimizes the use of state machine operations in situations where writers typically perform multiple updates of a data blob before other writers perform an update as follows. We mediate access to the metadata via single writer policies, and implement policy changes via the more expensive read-modify-write state machine operation. Simply reading and updating the metadata of a blob (by the current writer) is implemented using efficient distributed register operations. This approach ensures safe concurrent sharing of data blobs while minimizing the more expensive policy changes to cases when the writer for a blob changes. The full TEEMS interface is summarized in Table 4.1.

Storage operations involve the following sequence of steps. When an operation to write a new data blob d associated with id i is invoked, the client library starts by generating a symmetric encryption key k. In our implementation, we use an authenticated encryption scheme (AES-GCM), which generates a ciphertext ($\langle d \rangle_k$) and a MAC ($MAC(d)$) of d. The encrypted object and corresponding MAC are then stored in one or more untrusted cloud storage services, under a randomly created identifier i_{store}. Let a be the access control list for the newly stored data blob d. After the data has been successfully written to untrusted storage, the library contacts the TEEMS metadata service to update the metadata for id i:

$$\langle i, k, a, i_{store}, MAC(d) \rangle$$

The write operation concludes successfully after both the data write and the metadata update complete.

```
Write Metadata (key $k$, value $v$)
  1. smr_get_policy ($k$)
  2. if $eval(policy) ==$ ACCESS DENIED then
         return ACCESS DENIED
  3. register_write ($k$, $v$, $epoch$)
  4. if EPOCH CHANGED then
         restart operation
  5. return SUCCESS
```

Figure 4.1: Slow but trivially correct mixing of the protocols. The read operation is in all aspects similar.

Finally, the library deletes earlier versions of the blob from the data store.

When a read for id i is invoked, the client library starts by querying the TEEMS service for the most recent version of the metadata associated with the id i. After retrieving the tuple $\langle i, k_i, i_{store}, MAC(d_i)\rangle$, the client library then uses the id i_{store} to retrieve the encrypted data from untrusted storage. This encrypted data $\langle d' \rangle_{k_i}$ is read and then decrypted using k_i, obtaining d' and $MAC(d')$. Finally, its integrity is validated by comparing $MAC(d')$ and $MAC(d_i)$.

Finally, the access to the untrusted storage can be optimized by employing caching at the client. We can either cache full blobs (to avoid having to access the untrusted store) or name hints (for enabling parallel access to the metadata service and the untrusted store). In either case, the metadata always has to be fetched and compared with the cached or retrieved version, since only the TEEMS metadata service can ensure freshness of data blobs.

4.1.3 Leveraging different storage protocols

TEEMS implements the metadata read/write operations efficiently using our RR-tolerant ABD protocol (Section 3.3.2). However, updating the policies stored by TEEMS — which enables concurrent sharing — requires a read-modify-write operation, because changing a policy requires reading the policy first to check if the client has permission to modify it. Therefore, policy changes are implemented using the RR-tolerant SMR protocol (Section 3.3.3). This combination of protocols allows us to achieve both efficient reads and writes in the normal case, and concurrent write sharing via atomic policy changes through state machine updates.

In this design, the state of our state machine is an epoch number and an associated policy description. Crucially, the epoch number is also readable by the read/write protocol. As such, a policy change is a state machine operation which evaluates the current policy, replacing it with the proposed policy and incrementing the epoch number, if such permission is granted. The epoch number increment is then immediately visible to register operations.

For reads and writes, a slow but trivially correct combination of the protocols would be to issue a read operation on the state machine to obtain the policy and then issue the register operation, retrying if the

```
Write Metadata (key $k$, value $v$)
  1. smr_optimized_get_policy ($k$) in parallel with register_get_version ($k$)
  2. if $smr\_optimized\_get\_policy$ fails then
         fallback to slow operation
  3. if $eval(policy) ==$ ACCESS DENIED then
         return ACCESS DENIED
  4. broadcast WRITE REPLICA ($epoch$, $new\_ts$, $v$) and wait for $W_Q$ replies;
  5. return SUCCESS
```

Figure 4.2: Protocol for the fast intermixing of the distributed register and SMR protocols in TEEMS. By overlapping (and piggybacking) the optimized read to the state machine to get the policy and the reading of the current timestamp of the register (which is the first step of the write protocol), we can run both operations in parallel, since writing the value to the register only happens after the policy is verified. Note that the read operation can be similarly piggybacked (the value is only returned after the policy check)

epoch number has advanced, as described in Figure 4.1. The correctness of the combination hinges on the fact that 1) the policy is correctly read and enforced; and 2) by ensuring the epoch has not changed between checking the policy and operating on the register, the policy is guaranteed to be valid for that version of the object.

We note, however, that the initial phase of the register operation has the same communication pattern as the optimized read state machine operation. As such, it is possible to piggyback the optimized read request with the first phase of the register operation, as shown in Figure 4.2. If the fast policy read succeeds, the policy is evaluated and the operation proceeds. Otherwise, the system falls back on the slow path above. The optimization is correct because it is equivalent to the slower combination: the operation succeeds only if the policy is enforced and the register sub-operation only succeeds if it happens within the epoch of the policy.

The client side policy evaluation and protocol execution require trusted computation, meaning the implementation needs to either rely on a TEEMS replica as a proxy or on a client-operated TEE, attested by the replicas. In our description, as well as in our prototype implementation, we chose the former.

4.1.4 Security Properties

TEEMS ensures the confidentiality, integrity, and freshness of stored data blobs, due to the correctness of the underlying protocols, as discussed in Section 3.3.4. However, the untrusted storage is relied upon for availability of data blobs. For increased availability, clients may store multiple copies of a data blob (or erasure-coded fragments) at independent storage providers.

4.1.5 Deployment scenarios

To minimize the chance of correlated faults of individual metadata servers, each replica can be at a different location, depending on the deployment scenario. For example, the TEEMS replicas may be

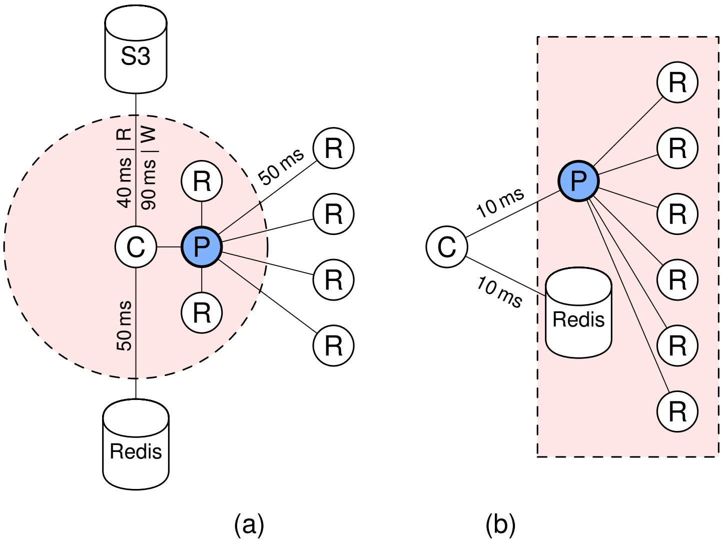


Figure 4.3: Deployment scenarios: (a) cloud client (with either Redis or S3), (b) collocation center with Redis. The shaded area represents a LAN (or a data center). Latencies will be used in the evaluation (in the case of S3, R represents the latency of reads and W the latency of writes). C is the client, R is a replica and P is the proxy replica.

distributed over multiple administrative domains to make it less likely that several of them can be rolled back at any given time. One way to achieve this distribution is to colocate some replicas with a client in a data-center, with the remaining replicas in other administrative domains (Figure 4.3a). In another example scenario, clients execute on their own premises and wish to share data items stored in the Cloud with other clients, without trusting the Cloud. They can use TEEMS replicas deployed at a local collocation center, where subsets can be physically isolated and operated by independent providers, possibly using storage in the same center (Figure 4.3b).

Depending on their cost and availability needs, clients may opt to store a single copy on a single cloud storage service provider, multiple copies on independent providers, or multiple erasure coded fragments on independent providers. In the common case, a copy or a small set of fragments can be efficiently retrieved from the nearest providers.

4.2 Implementation

We implemented the TEEMS prototype in Intel SGX (version 1), using C++. The TEEMS servers are implemented using a single-threaded event loop and 6KLoC (TEEMS). Additionally, the client library, which interacts with the replicas and with the untrusted storage, takes up to 5KLoC (TEEMS). We used Flatbuffers [88] to define our protocol messages and their serialization, but implemented the secure transport between the replicas using OpenSSL [89].

It is important to observe that since in TEE replicated systems replicas cannot generally trust clients to execute the protocol code correctly, our prototypes implement the driver code collocated with the replicas. This means that, for instance, a register write request is first sent to a TEEMS server (typically, one collocated with the client) which acts as a proxy and runs the two rounds of the protocol. A possible alternative would have been to implement a TEE library that drives the register protocol, which all replicas would remotely attest. We chose not to do this, since the former approach has wider applicability (e.g., it allows clients that do not run in TEE-enabled platforms to interact with a TEE replicated system).

All prototypes are open-sourced under the MIT license and available at `https://gitlab.mpi-sws.org/restart-rollback`.

4.3 Evaluation

We evaluate the TEEMS prototype based on the RR model using benchmark workloads, aiming to answer the following questions:

1. What is the overhead added by TEEMS to secure a cloud storage system under different deployments? (§4.3.1)
2. What is the performance of this system when used to store metadata for a KVS (Redis) under a benchmark workload (YCSB)? (§4.3.2)

Experimental Setup. We ran our experiments using 7 machines with Intel® Xeon® E-2174G processors running Debian Linux version 4.19 to run the replicas, plus 2 machines equipped with Intel® Xeon® Platinum 8260M processors running Debian Linux version 5.4: one of them to execute the clients and the other to host Redis. We again leveraged `sloth` [85] to simulate different network topologies.

In our result graphs, each data point represents the median measurement over $3,000$ requests and the error bars show the 5^{th} and 95^{th} percentiles.

4.3.1 TEEMS-based storage

First, we focus on understanding the performance of our example application, the TEEMS-based secure storage service, under different deployments, which differ in the relative location of the client, the metadata servers, and the type and location of the untrusted storage. In this set of experiments, our baselines are the untrusted storage systems being used in each situation, namely Amazon S3 and an instantiation of Redis on our local cluster, to which we optionally add a variable latency on the access link.

We consider three representative deployment scenarios, illustrated in Figure 4.3.

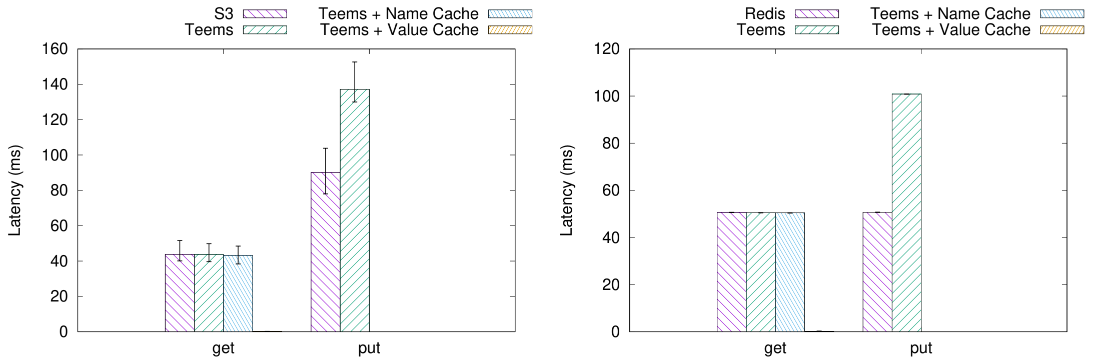


Figure 4.4: Client in the Cloud w/ S3

Figure 4.5: Client in the Cloud w/ Redis

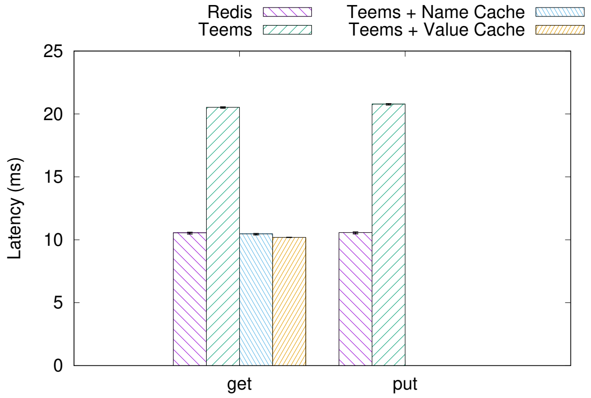


Figure 4.6: Collocation Center w/ Redis

Figure 4.7: Latency in different deployment scenarios. The "s3" and "redis" bars refer to direct accesses to the untrusted storage, providing a performance baseline without rollback protection. "teems" refers to accesses without caching. "teems + name cache" and "teems + blob cache" refer to accesses where the name and blob caches had a hit, respectively. Caching only applies for get requests, and in the cases of Figures 4.4 and 4.5 the "value cache" exists, but is close to zero since the cache hit with a local read quorum means there is no network latency access in the critical path.

Client in the Cloud. In this deployment (Figure 4.3a), the client is co-located with three metadata servers and the remaining four servers are in another data-center. We use two variants of storage: a remote Redis deployment, and S3.

Collocation Center. In this scenario (Figure 4.3b), all metadata servers and the Redis deployment are in the same collocation center, being hosted by different cloud providers.

In both cases, the largest administrative domain has 4 replicas, while at most 2 replicas are expected to crash in a correlated fashion. As such, we consider $M_R = 4$ and $F = 2$.

From Figures 4.4–4.6, we conclude that: 1) TEEMS performance depends heavily on the deployment scenario (in particular on the existence of local read quorums, which are enabled by RR); 2) name caching is effective at masking the overhead of accessing the metadata store; 3) blob caching, when combined with local read quorums, allows for local reads, outperforming the baseline.

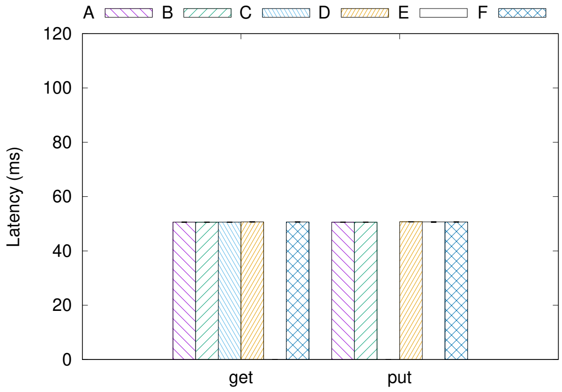


Figure 4.8: Redis baseline

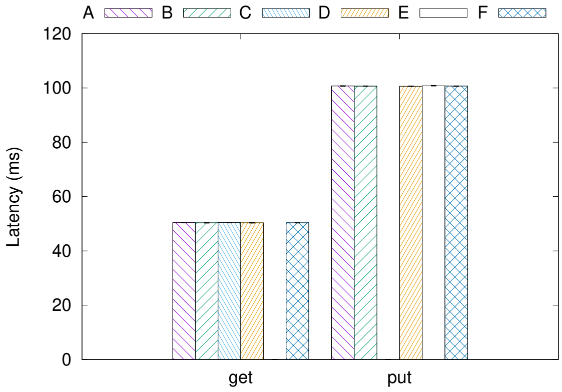


Figure 4.9: TEEMS + Redis

Figure 4.10: Latency with YCSB workloads

4.3.2 TEEMS-based storage running YCSB

Next, we compare TEEMS with using only Redis, on the YCSB benchmark workloads. We deployed TEEMS in the Client in the Cloud setting with Redis, using name caches. We use all six core workloads of YCSB (A–F), which have different key distributions and read/write ratios. The size of the objects varies between 100B and 1KB, depending on the workload and the overall size of the database is of 1000 1KB objects. The results in Figures 4.9 and 4.8 show that writes incur a $2\times$ overhead, which is expected since the Redis access must be preceded by the TEEMS metadata access. In contrast, reads perform comparably to the baseline, due to local read quorums and effective usage of the cache.

5

Leveraging the RR model in distributed storage

Contents

5.1 Motivation 55
5.2 R2-S2 Design 56
5.3 Parameterizing the System 61
5.4 Preliminary Evaluation 63

In this chapter, we will present the Restart-Rollback Storage System (R2-S2), a replicated KVS based on LevelDB [55], a single node KVS implemented using LSM-trees [90]. We have developed R2-S2 to showcase the broad applicability of the RR model by using it in a non-security context, without TEEs.

The motivation for R2-S2 is presented in Section 5.1, followed by its design in Section 5.2. We present guidelines for system parameterization in Sections 5.3. A preliminary evaluation is presented in Section 5.4.

5.1 Motivation

Batching multiple small write operations to a block-oriented storage device into a single large one is a known mechanism to increase its throughput. To achieve this batching, one has to wait for an adequate amount of contiguous data to write, which introduces a delay in the operation. Alternatively, once the data to write is present in an in-memory buffer (soft state) the system could return to the client. This opens the possibility of data loss, if the system crashes before effectively synchronizing this data to the persistent storage device.

This problem is propagated as is to current persistent replicated storage systems: the throughput of the system is limited by the throughput of the storage devices of the replicas. To avoid this performance penalty, some replicated systems consider data to be persisted if it is present in volatile memory at a sufficiently large subset of replicas [21] (i.e., persistence through replication). Such an approach is not consensual, with other authors insisting that data must be present in stable storage to be considered persisted [3].

We argue that these two properties — performance and durability — do not need to be mutually exclusive. Instead, we aim to combine both into a system that offers the throughput from the solutions that persist in the background with the high durability from systems that persist before replying to requests.

To this end, we present a novel replication strategy based on carefully and strategically mixing both approaches, leveraging RR quorum systems. Instead of issuing the write requests symmetrically to all replicas, as is customary, we can strategically signal to certain replicas to synchronize to persistent storage before replying while allowing others to acknowledge the write eagerly, thus mixing both approaches. This key insight allows us to take advantage of batching, improving the overall performance of the system, without relinquishing fault tolerance. The RR fault model captures perfectly replicas that crash before synchronizing their volatile batches to stable storage. In this case, these replicas suffer a rollback, but since they have restarted can flag their suspicion regarding the freshness of their state.

This approach opens a series of challenges. The storage layer requires careful adaptations to support multiple modes of operation while offering efficient batching, even on non-sequential writes. At the replication level, the asymmetry of operation between replicas introduces the problem of sync scheduling: choosing which replicas synchronize to persistent storage and which can reply eagerly. This schedule is paramount to extracting the maximum performance of the system, and offers a rich design space. Parameterizing a deployment of such a system is also not obvious. There is a tradeoff to be explored between the total number of replicas, the number of replicas that synchronize to disk in the background (and, by extension, the performance of the system) and the desired availability and reliability of the system. We present a principled method for this parameterization, where a system administrator can tune the number of replicas based on how prone to faults they are and the desired availability and durability levels.

5.2 R2-S2 Design

This section presents the design of R2-S2 based on the principle of asymmetric synchronization. However, before discussing our architecture in detail, it is worthwhile to categorize in a systematic way the three approaches that can be used to synchronize replica state to stable storage.

Sync immediately. When receiving an object to write, a replica immediately writes and flushes the object to stable storage, issuing the reply afterwards. This prevents batching, which favours durability and latency in detriment of throughput.

Batch and wait. When receiving an object to write, a replica places the object in a batch and waits for this batch to be flushed to stable storage. Afterwards, the reply can be issued. Since there is batching and objects are being persisted before replying, this approach favours durability and throughput at the cost of latency (which can be very large if the batch takes a long time to be filled due to a lack of writes).

Batch and reply (Move fast and break things). In this approach, replicas place the object in the batch (i.e., volatile memory) and reply immediately. This achieves the best latency and throughput, sacrificing durability since replicas can suffer a rollback if they crash and restart.

The state of the art seems to hint at an impossibility trinity: systems have to pick two between latency, throughput, and durability. The design goals of R2-S2 aim to question this impossibility, by showing that there can be interesting combinations of these features that, at least, approximate the best of the three approaches: R2-S2:

1. **Strong Persistence**: data loss should never occurr, even in the presence of arbitrary node crashes;
2. **Performance**: the architecture should yield performance gains compared to Approaches A and B;

Note that none of our baselines satisfy all requirements: approaches A and B achieve strong persistence but not the performance requirement. Approach C does meet the performance requirement, but does not satisfy strong persistence.

In the remainder of this section, we will describe asymmetric synchronization, the key technique that realizes the goals prescribed above, and how the architecture needs to accomodate this paradigm to achieve the best possible performance.

5.2.1 Asymmetric Synchronization

Asymmetric synchronization is the key to extracting performance in R2-S2. The core idea is to partition the replica set into two disjoint subsets: the volatile set $\mathbb{R}$, composed of the replicas that eagerly reply to the request before synchronizing to persistent storage (and thus can suffer a rollback on restart); and the sync set $\mathbb{F}$, comprised by the replicas that wait until the data is persisted before replying. When issuing a write request, each replica is told in which set they belong to, and acts accordingly. An important subset of $\mathbb{F}$ is its intersection with the write quorum $\mathbb{W}_Q$. This subset is comprised by the replicas that are critical to the performance of the write operation (in particular, its latency) and is referred as the critical sync set from now on.

Achieving strong persistence. To achieve strong persistence, it is sufficient that the critical sync set is never null. This implies that there is always at least one replica in every write quorum which persisted the value, making it recoverable. This is guaranteed by the RR quorum system, by setting the maximum size of $\mathbb{R}$ to M_R, since the volatile replicas are the ones which can suffer the rollback on restart.

Improving performance. The key to improving throughput (compared to the sync immediately baseline) is scheduling. Intuitively, this will distribute the burden of syncing to disk among the replica set, allowing all the replicas to batch eventually. By having all replicas reply as soon as possible we also get a latency improvement compared to the batch and wait baseline. Regretablly, although these approaches does yield performance benefits compared to sync immediately and batch and wait, it is impossible to match the performance of the batch and reply baseline: this would require removing any synchronizing writes from the critical path, making $\mathbb{W}_Q \cap \mathbb{F} = \varnothing$, violating strong persistence.

5.2.2 System Overview

The R2-S2 system is comprised of a client and several servers, where the client is implemented as a library that implements the replication layer and handles the connection to and coordination of the

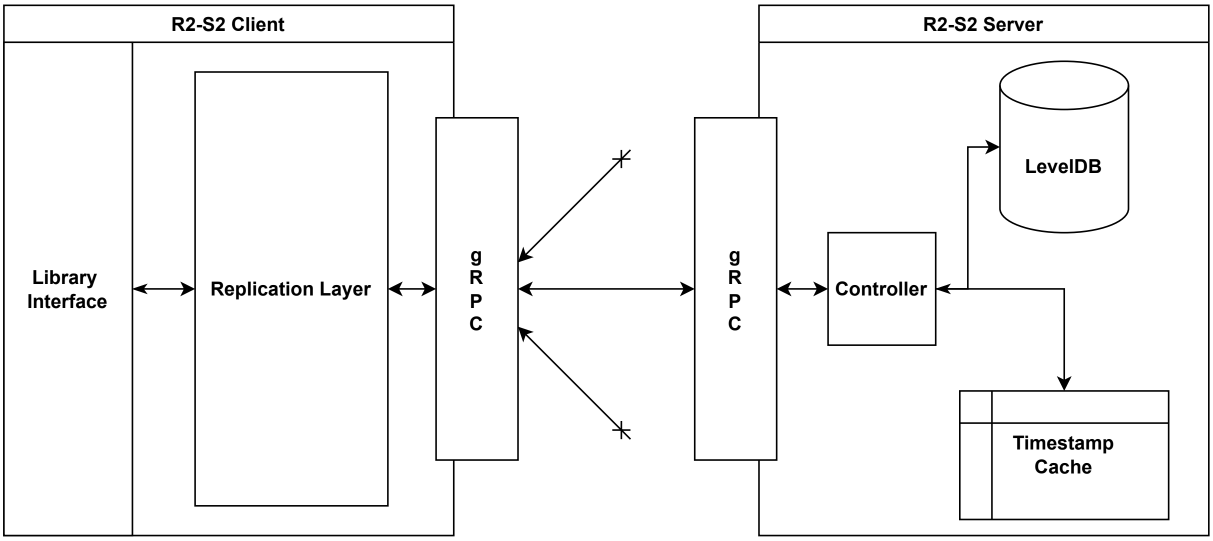


Figure 5.1: Architecture of the R2-S2 prototype

multiple R2-S2 servers. This replication layer also decides and coordinates the replicas to implement the schedule (which replicas have to synchronize the write before replying).

Each R2-S2 server consists of an RPC controller, which handles the interface with the client. This controller manages the server's local KVS (in our case, LevelDB [55]). In our prototype, we chose to implement the ABD [1] protocol, since its read/write interface maches perfectly with the KVS interface we are aiming to implement.

One relevant aspect of ABD, and most other replication protocols, is the notion of object versions, which are not directly supported by the KVS interface. Although we store the versions with the objects for persistence, in order to order concurrent writes to the same object with different timestamps, an in memory timestamp cache is required. This cache is populated as needed, with synchronization on the values to order the writes to ensure that a newer value is not overwritten by a previous one. Additionally, this cache enables faster queries of the object's version, which is the first phase of an ABD write.

The overall architecture of the R2-S2 prototype is summarized in Figure 5.1.

5.2.3 Scheduling

As we have covered, a good schedule is crucial to realize the performance potential of the architecture. A schedule is a sequence S, where S_i^j is the mode of operation (batch and reply, batch and wait or sync immediately) to be applied by replica i for the j^{th} write operation. The schedule is said to be valid if the volatile set (i.e, the number of batch and reply modes of operation) does not exceed M_R, for every S^j. Thus, any valid schedule guarantees strong persistence.

There are several approaches to scheduling. Which is preferable is non-obvious beforehand: the

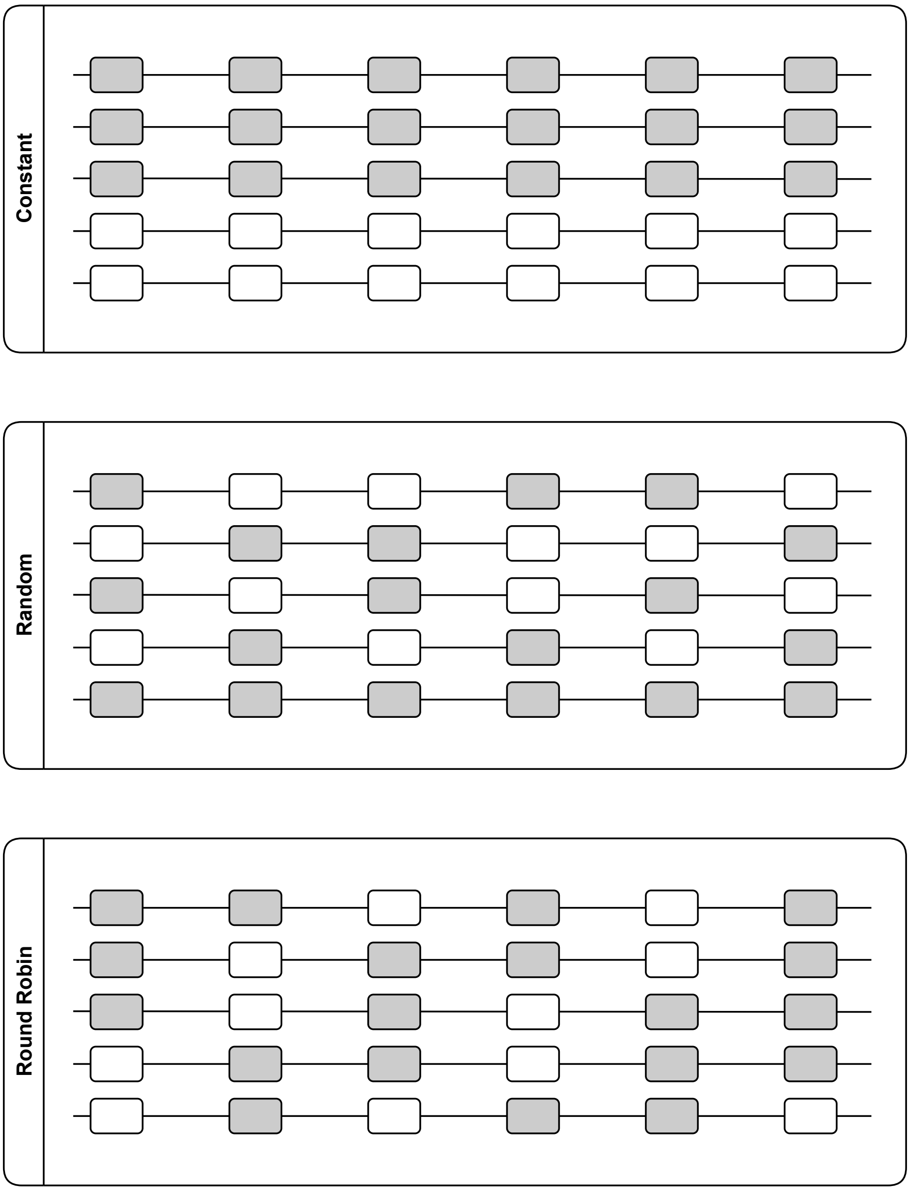


Figure 5.2: Examples of valid static schedules for a parameterization of 5 nodes with $M_R = 2$. In each schedule a line represents a replica and the temporal sequence of objects it writes (boxes). A shaded box represents an object that was synchronized immediately, while a white box represents a batched object.

schedule will be sensitive to the particularities of the replication protocol, the deployment and the operation load. In the preliminary evaluation in § 5.4 we present an evaluation of some of the scheduling policies described here.

At a high level, there are two types of schedules: static schedules, which can predetermined ahead of time, and as such are independent of the load the system is being subjected to; and dynamic schedules, which are computed on the fly and as such can be reactive to the load and current status of the system. In this dissertation we will focus on static schedules, as they are simpler and sufficient to showcase the gains from the RR model.

Constant Schedule. A constant schedule is one where all write operations use the same modes of operation for each replica. In other words, all S^js are equal.

Random Schedule. A schedule where replicas are randomly allocated to the sync and volatile sets (based on the sizes they need to have for the schedule to be valid).

Round Robin Schedule. A schedule where the sync set is shifted at each round. For instance, in a system with $N = 5, M_R = 2$, replicas $0, 1, 2$ are the sync set at step j, and at step $j + 1$ replicas $3, 4, 0$ become the sync set, and so on.

Figure 5.2 shows examples of different static schedules.

5.2.4 Storage Layer

The storage layer plays a crucial role in realizing the performance potential of the system. It needs to efficiently support three different types of write operations, one for each of the different baseline approaches. It will also need to support read operations efficiently. Figure 5.3 shows a diagram of the storage layer and its data-structures.

An LSM-tree is employed as the core data structure of the storage layer, enabling the sequential writes required for batching as well as providing reasonable read performance. There exists a log, used to write values that need to be synced immediately, and an in-memory batch for values that should be batched. To further speed up reads there is an optional in-memory object cache.

There is an additional in-memory cache, which only stores the object versions. This is useful because versions are a staple of most replication protocols and several of them require reading the current version before writing a new one. Since versions are constant-sized and small, the capacity of the version cache can be significantly larger than that of the object cache. This cache is required to synchronize concurrent writes of different versions to the same key. When the node is writing a value, it required to check the version cache, populating it if required, where an atomic update is executed on the version to ensure that an old version does not override a newer version it is racing with.

With these data structures in place, the various operations requried to implement the KVS interface and asymmetric syncing become quite simple. Satisfying version reads is simple: the timestamp cache is consulted (falling back on the LSM-tree and log if needed). The workflow of an object read is similar: if the object cache exists it is consulted, falling back on the LSM-tree and the log in the event of a miss. Handling batch and reply writes requires placing the value in the batch and eventually a background thread will flush the batch into the LSM-tree. Handling sync immediately writes amounts to placing it

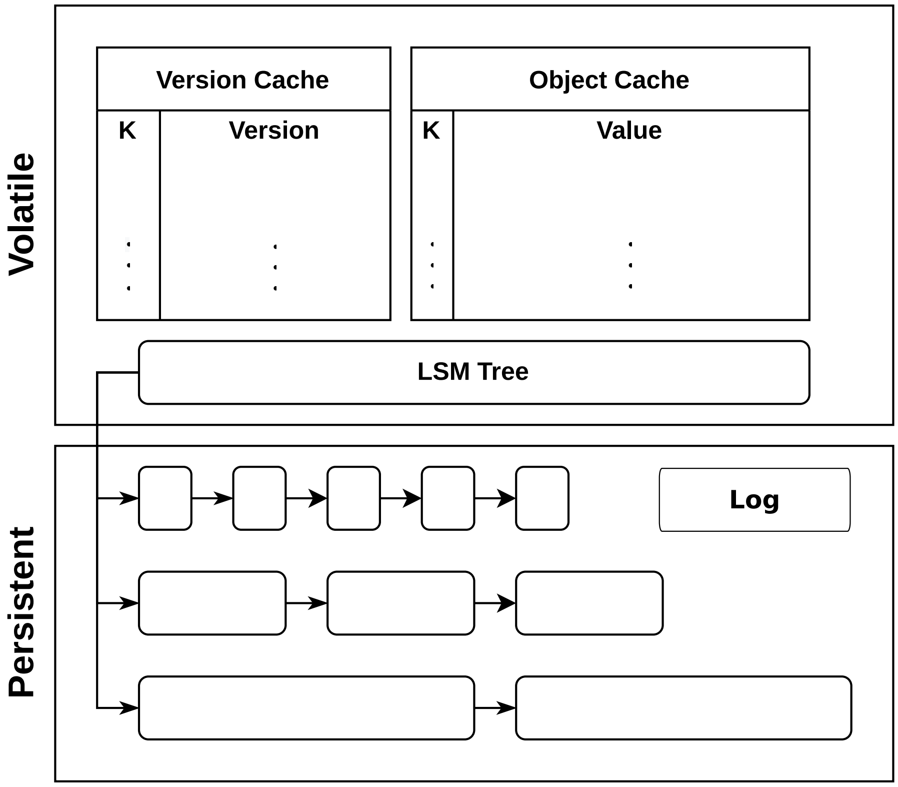


Figure 5.3: Diagram of the Storage Layer

in the persistent log, with this log being compacted into the LSM-tree at a later time by a background thread.

5.3 Parameterizing the System

To parameterize our system, we effectively need to choose appropriate values for M_R and F. We employ a data-driven approach to this problem. Relying on real-world information about the availability and reliability of components, we can predict the reliability and availability of the system as a function of M_R and F. Choosing the adequate values becomes an exercise of setting the desired reliability/availability and solving for M_R and F.

The availability of the system (probability that it is available at a particular point in time) is impacted by the availability of the individual replicas. Let p_c be the symmetric of the availability of a replica, ie: the probability that a replica is crashed at a particular point in time.

The reliability (probability that the system loses data over a given period of time) of the system is dependent on the reliability of the underlying storage devices and the availability of the replicas in the eager set. Let p_f be the probability that a storage device fails (irrecoverably) in that period of time.

Note that generally, the probability of a replica crashing is higher than that of the replica's underlying storage failing. As such, in this discussion we assume that $p_c > p_f$.

In order to derive the parameterization of the system, let us begin with a simplifying assumption that all failures (both crashes and storage device failures) are independent among all replicas, an assumption valid for a geo-replicated deployment, for instance. Let us first consider reliability. The probability that all replicas in the eager set crash (and thus all the replicas that only had the data in volatile storage lose that data) is $p_c^{M_R}$, since M_R is the size of the eager set. To truly lose the data, the remainder of the replicas in the write quorum would have had to suffer an irrecoverable fault in their storage devices, which happens with probability $p^{W_Q - M_R}$. Due to the independence of failures, we can derive the probability that data is lost to be $p_c^{M_R} \cdot p_f^{W_Q - M_R}$, making the reliability of the system be $1 - p_c^{M_R} \cdot p_f^{W_Q - M_R}$. Considering availability, we know that the system is available if a read quorum and a write quorum are available. Due to the independence of failures, the quorums become unavailable with probability $p_c^{R_Q}$ and $p_c^{W_Q}$, respectively. As such, the probability that they are both available at a given point in time is simply $1 - p_c^{\min(R_Q, W_Q)}$.

Now let us imagine that the probability of replicas crashing is correlated, as is the case in a data-center deployment. Under this assumption, an event like a power outage will cause all replicas to crash and lose their volatile state. Therefore, the probability of n replicas crashing is the same as the probability of a single replica crashing: p_c. Considering reliability, this means that the probability of the eager set losing its volatile state due to a crash and the remainder of the write quorum suffering an irrecoverable failure is $p_c \cdot p_f^{W_Q - M_R}$. Availability is quite simpler to derive, as if the probability of quorums of any type being unavailable is simply p_c, making the availability of the system $1 - p_c$.

Table 5.1 summarizes the availability and reliability of the system in these two scenarios.

Table 5.1: Availability and reliability in a geo-replicated deployment (uncorrelated failures) versus a data-center deployment (correlated failures). R_Q and W_Q are the sizes of the quorums for a parameterization with M_R tolerated rollbacks and F crash faults

	Geo-replicated	**Datacenter**
Availability	$1 - p_c^{\min(R_Q, W_Q)}$	$1 - p_c$
Reliability	$1 - p_c^{M_R} \cdot p_f^{W_Q - M_R}$	$1 - p_c \cdot p_f^{W_Q - M_R}$

It is worthwhile to compare the durability guarantees of R2-S2 with the CFT baselines where all replicas either sync before replying (i.e., batch and wait and sync immediatelly) or batch and reply.

In the former case, since when replies are issued the updates are persisted on all replicas, the reliability is independent of the existence of crash faults (and, consequently, of the deployment scenario). The probability of data being lost is therefore p_f^Q, where Q is the size of a quorum, making the reliability $1 - p_f^Q$, which is significantly higher than that provided by R2-S2 in either deployment scenario. This equates to this baseline requiring less replicas to achieve the same level of reliability.

In the latter case, in which all replicas use the batch and wait approach, the reliability can be derived from the probability of a quorum crashing, since it is not guaranteed that the quorum persisted the value before replying. Therefore, the probability of data loss is deployment sensitive. In the geo-replicated scenario, the probability of a quorum crashing is p_c^Q, whereas in the data-center scenario (where crash faults are correlated) this probability is simply p_c, resulting in a reliability of $1-p_c^Q$ and $1-p_c$, respectively. As expected, these values are lower than those offered by R2-S2. Moreover, in the data-center scenario the reliability is independent of the number of replicas, since they are likely to crash simultaneously.

5.4 Preliminary Evaluation

In this section, we present our preliminary evaluation of the R2-S2 prototype. We aim to answer two fundamental questions about the approach:

1. What performance benefits can be extracted using asymmetric synchronization? (§5.4.3)
2. What is the impact of different schedules in the performance of the R2-S2 system? (§5.4.4)

5.4.1 Implementation

We implemented R2-S2 using LevelDB [55] as the underlying storage layer and our own replication layer on top of it. This was extremely helpful since LevelDB already implements the LSM-tree, the persistent log and the optional object cache out of the box, and provides for two out of the three modes of operation: batch and reply and sync immediately. We intend to implement the batch and wait mode of operation and incorporate it in our prototype. However, this requires changing LevelDB to track the batches each object is placed in and when those batches are eventually persisted.

The replication layer was implemented in Rust with the request transport being handled by gRPC [91]. The R2-S2 client library provides a shim that interacts with the replication layer that coordinates the ABD operations. The R2-S2 server receives requests via gRPC and manages the timestamp cache and its embedded LevelDB instance.

The client library was implemented in approximately 2KLoC of Rust. The server was implemented in 400LoC. The gRPC protocol definition is comprised of 135 LoC. All the remainder driver code for benchmarks consists of 2KLoC of Rust.

All prototypes are open-sourced under the MIT license and available at `https://gitlab.mpi-sws.org/restart-rollback`.

5.4.2 Methodology

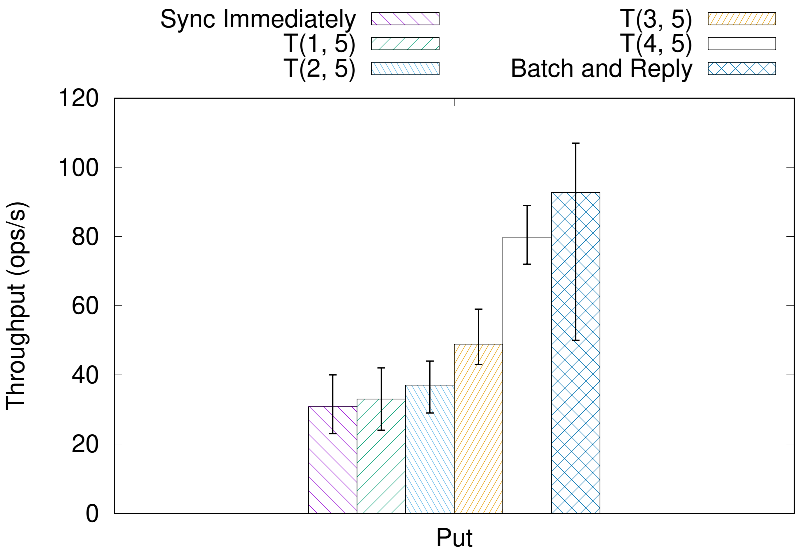


Figure 5.4: Maximum throughput for different configurations. The batch and reply configuration places the object in a batch and replies immediately, while in the sync immediately configuration all replicas flush the object to the persistent log before replying. In the $T(BR, 5)$, BR replicas in each write use the former approach, while the remaining use the latter.

Experimental Setup. We ran our experiments using 5 machines, each with an Intel® Xeon® E-2174G processor and a 300GB Seagate Magnetic disk model ST300MM0006, running Debian Linux version 4.19 to run the replicas, plus one machine equipped with a Intel® Xeon® Platinum 8260M processor running Debian Linux version 5.4 to execute the client driver of the benchmarks.

Experimental Methodology. In our result graphs, each data point represents the average measurement throughput measurement over 60 seconds of running at load, using an open-loop client (i.e., a client that issues a request asynchronously at regular time intervals to achieve the desired load). To find the throughput in each configuration, we ran the experiment with multiple offered loads to find the maximum throughput point.

5.4.3 Asymmetric Synchronization

In this experiment, we compare two baselines in the classical CFT model, sync immediately and batch and reply, with a set of threshold configurations ($T(BR, N)$), which vary the number of replicas that use the batch and reply synchronization mode. Effectively, $T(0, N)$ is the CFT sync immediately configuration, which corresponds to the classic system where each replica persists before replying. Moreover $T(N, N)$ is the CFT batch and reply configuration, which only offers guarantees of the persistence through replication group, since the client obtains a reply to the write operation before data is guaranteed to be persisted in any of the replicas. Configurations $T(1, 5)$ and $T(2, 5)$ offer the same persistence

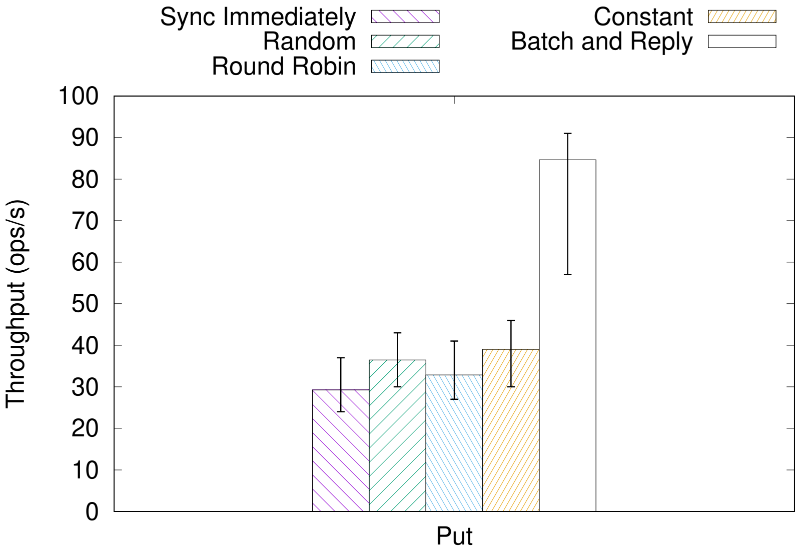


Figure 5.5: Maximum throughput for different schedules. The batch and reply configuration places the object in a batch and replies immediately, while in the sync immediately configuration all replicas flush the object to the persistent log before replying. The constant schedule always synchronizes to the same 3 of the 5 replicas. The randomized schedule also synchronizes to 3 replicas, but these get randomized every operation. The round robin schedule changes the 3 replicas at each operation in a cycle.

guarantees of $T(0,5)$, while $T(3,5)$ and $T(4,5)$ offer a more subtle compromise. In these configurations there are enough replicas in the batch and reply mode to form a write quorum. This means that in the event of a full system shutdown, if the other replicas that are in fact persisting the write have not yet processed the request (or the message was lost) the data will be lost. Nevertheless, these (probabilistic) durability guarantees are still qualitatively better than those of the batch and reply baseline.

We measured the maximum throughput of each configuration using a randomized schedule and $128KB$ objects. As we can observe in Figure 5.4, the more replicas are allowed to batch the higher the average throughput. We also note the sharp increase in throughput the $T(3,5)$ and $T(4,5)$ configurations. This is for precisely the same reason as their poorer durability guarantees: the batch and reply replicas are enough to form a write quorum, making the throughput of the overall system significantly better.

5.4.4 Scheduling

Next, we compare the same two baselines in the CFT model, sync immediately and batch and reply with three RR based quorum configuration which aim to combine the durability of the former and performance of the latter. Each of the RR configurations uses a different static schedule, as described in Section 5.2.3: constant, random and round robin. These configurations have $M_R = 2$ and $F = 2$. This matches both the number of tolerated crash faults of the CFT baselines and the total number of replicas.

We measured the maximum throughput of each configuration using $128KB$ objects. As we can observe in Figure 5.5, all RR configurations achieve throughput gains of up to 33% when compared to the sync immediately baseline, while still achieving strong persistence. However, there is still a large gap to the batch and reply baseline. Even though part of this gap is fundamentally attributed to the lower durability guarantees offered by the batch and reply configuration, we also intend to bridge this implementation gap and improve these results.

6

Conclusions and Future Work

In this dissertation, we have presented the Restart-Rollback (RR) fault model, designed for replicated TEEs with external state. We have provided an extensive formalization of the model, resulting quorum systems and its properties, and laid out guidelines for adapting existing CFT protocols to the model with little effort. We have furthermore experimentally verified that these resulting protocols are on par in terms of performance with their CFT counterparts (which do not provide safety in the case of replicated TEEs with persistent state) and significantly outperform the BFT counterparts that would otherwise be required to tolerate rollback faults.

We have shown the general applicability of this model, both in TEEMS and R2-S2. In TEEMS, we have designed and implemented a metadata service that can be used to enhance existing cloud storage systems with TEE-grade security. Our evaluation of TEEMS shows that the overhead it incurs is modest. In R2-S2, we have shown the potential of this fault model outside the TEE use case, by designing a distributed KVS that strikes a balance between performance and durability, which would otherwise be impossible in the CFT model. Our preliminary evaluation of R2-S2 shows that there are performance gains in using the RR model, compared to simply synchronizing to all replicas.

In the future, we aim to refine the R2-S2 prototype, showing its applicability in other replication protocols other than ABD. We also wish to modify LevelDB to be able to implement the batch and wait semantics, and further improve the performance of the system. Additionally, we aim to conduct a survey on frequency of failures (both crash faults and disk corruptions) that would help us instantiate the parameterizations described in Section 5.3 with real world data. Another high level forward looking vision is to study the possible benefits and limitations of conceptually separating a node's computation (code and volatile memory) and its persistent state. This approach would not only provide an interesting description of the RR model but also have wider applicability, for instance in the context of network connected disks that can be used by multiple nodes.

Bibliography

[1] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," in *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '90, 1990, p. 363–375.

[2] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169. Also appeared as SRC Research Report 49. This paper was first submitted in 1990, setting a personal record for publication delay that has since been broken by [60].*, May 1998, aCM SIGOPS Hall of Fame Award in 2012. [Online]. Available: https://www.microsoft.com/en-us/research/publication/part-time-parliament/

[3] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, "Paxos replicated state machines as the basis of a {High-Performance} data store," in *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[4] ARM, "ARM security technology. Building a secure system using trustzone technology," White paper PRD29-GENC-009492C, 2009.

[5] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," White paper at https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2016, accessed: 2020-05-24.

[6] AMD, "AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More," White paper at https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, 2020, accessed: 2020-05-27.

[7] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 86, 2016.

[8] Intel, "Intel trust domain extensions," White paper PRD29-GENC-009492C, 2020.

[9] R. Grisenthwaite, "Arm cca will put confidential compute in the hands of every developer," https://www.arm.com/company/news/2021/06/

arm-cca-will-put-confidential-compute-in-the-hands-of-every-developer, Jun. 2021, accessed: 2020-12-14.

[10] J. Lind, O. Naor, I. Eyal, F. Kelbert, E. G. Sirer, and P. Pietzuch, “Teechain: A secure payment network with asynchronous blockchain access,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 63–79. [Online]. Available: https://doi.org/10.1145/3341301.3359627

[11] S. Matetic, A. Mansoor, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, “ROTE: Rollback protection for trusted execution,” in *Proc. of the 26th Usenix Security Symposium*, 2017.

[12] M. Bailleu, D. Giantsidi, V. Gavrielatos, D. L. Quoc, V. Nagarajan, and P. Bhatotia, “Avocado: A secure In-Memory distributed storage system,” in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 65–79.

[13] Y. Jia, S. Tople, T. Moataz, D. Gong, P. Saxena, and Z. Liang, “Robust p2p primitives using sgx enclaves,” in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, 2020, pp. 1185–1186.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with ycsb,” in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC ’10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[15] M. Larrea, A. Fernández, and S. Arévalo, “On the impossibility of implementing perpetual failure detectors in partially synchronous systems,” in *Proceedings 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*. IEEE, 2002, pp. 99–105.

[16] B. Awerbuch, Y. Mansour, and N. Shavit, “Polynomial end-to-end communication,” MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE, Tech. Rep., 1989.

[17] R. Guerraoui, M. Hurfinn, A. Mostéfaoui, R. Oliveira, M. Raynal, and A. Schiper, “Consensus in asynchronous distributed systems: A concise guided tour,” in *Advances in Distributed Systems*. Springer, 2000, pp. 33–47.

[18] D. Skeen and M. Stonebraker, “A formal model of crash recovery in a distributed system,” *IEEE Transactions on Software Engineering*, no. 3, pp. 219–228, 1983.

[19] L. Lamport, R. Shostak, and M. Pease, “The byzantine generals problem,” *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, p. 382–401, Jul. 1982.

[20] D. Malkhi and M. Reiter, “Byzantine quorum systems,” in *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, ser. STOC ’97, 1997, p. 569–578.

[21] M. Castro and B. Liskov, “Practical byzantine fault tolerance,” in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI ’99. USENIX Association, 1999, p. 173–186.

[22] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, “Upright cluster services,” in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP ’09, 2009, p. 277–290.

[23] F. J. Meyer and D. K. Pradhan, “Consensus with dual failure modes,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 2, p. 214–222, Apr. 1991. [Online]. Available: https://doi.org/10.1109/71.89066

[24] M. Backes and C. Cachin, “Reliable broadcast in a computational hybrid model with byzantine faults, crashes, and recoveries,” in *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, 2003, pp. 37–46.

[25] M. Correia, L. C. Lung, N. F. Neves, and P. Verissimo, “Efficient byzantine-resilient reliable multicast on a hybrid failure model,” in *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, 2002, pp. 2–11.

[26] R. Guerraoui and M. Vukolic, “Refined quorum systems,” in *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC ’07, 2007, p. 119–128.

[27] H. Howard, D. Malkhi, and A. Spiegelman, “Flexible Paxos: Quorum Intersection Revisited,” in *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, 2016, pp. 25:1–25:14.

[28] D. Porto, J. a. Leitão, C. Li, A. Clement, A. Kate, F. Junqueira, and R. Rodrigues, “Visigoth fault tolerance,” in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys ’15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2741948.2741979

[29] F. B. Schneider, “Implementing fault-tolerant services using the state machine approach: A tutorial,” *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[30] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty process,” *J. ACM*, vol. 32, no. 2, p. 374–382, Apr. 1985.

[31] L. Lamport, “Fast paxos,” *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[32] I. Moraru, D. G. Andersen, and M. Kaminsky, “Egalitarian paxos,” in *ACM Symposium on Operating Systems Principles*, 2012.

[33] E. Gafni and L. Lamport, “Disk paxos,” *Distributed Computing*, vol. 16, no. 1, pp. 1–20, 2003.

[34] J. Kirsch and Y. Amir, “Paxos for system builders: An overview,” in *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, ser. LADIS ’08, 2008.

[35] D. Ongaro and J. Ousterhout, “The raft consensus algorithm,” 2015.

[36] A. Baumann, M. Peinado, and G. Hunt, “Shielding applications from an untrusted cloud with haven,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, Oct. 2014, pp. 267–283.

[37] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, “Vc3: Trustworthy data analytics in the cloud using sgx,” in *36th IEEE Symposium on Security and Privacy, S&P 2015*, May 2015.

[38] M. Russinovich, “Introducing Azure confidential computing,” https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/, 2017, accessed: 2020-05-19.

[39] Google, “Confidential vm and compute engine,” https://cloud.google.com/compute/confidential-vm/docs/about-cvm, 2021, accessed: 2021-05-049.

[40] A. K. Reddy, P. Paramasivam, and P. B. Vemula, “Mobile secure data protection using eMMC RPMB partition,” in *2015 International Conference on Computing and Network Communications (CoCoNet)*, 2015, pp. 946–950.

[41] B. Parno, J. Lorch, J. Douceur, J. Mickens, and J. McCune, “Memoir: Practical state continuity for protected modules,” in *32nd IEEE Symposium on Security and Privacy, S&P 2011*, May 2011, pp. 379–394.

[42] “TPM main specification level 2.” Trusted Computing Group. Version 1.2, Revision 116., Mar. 2011.

[43] R. Strackx, B. Jacobs, and F. Piessens, “Ice: a passive, high-speed, state-continuity scheme,” in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC ’14, Dec. 2014, pp. 106–115.

[44] R. Strackx and F. Piessens, “Ariadne: A minimal approach to state continuity,” in *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, Aug. 2016, pp. 875–892.

[45] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, “Attested append-only memory: Making adversaries stick to their word,” in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP ’07, 2007, p. 189–204.

[46] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "Trinc: Small trusted hardware for large distributed systems," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'09. USENIX Association, 2009, p. 1–14.

[47] M. Brandenburger, C. Cachin, and N. Knežević, "Don't trust the cloud, verify: Integrity and consistency for cloud object stores," *ACM Transactions on Privacy and Security (TOPS)*, vol. 20, no. 3, pp. 1–30, 2017.

[48] C. Cachin, I. Keidar, and A. Shraer, "Fail-aware untrusted storage," *SIAM Journal on Computing*, vol. 40, no. 2, pp. 493–533, 2011.

[49] C. Cachin and O. Ohrimenko, "Verifying the consistency of remote untrusted services with commutative operations," in *International Conference on Principles of Distributed Systems*. Springer, 2014, pp. 1–16.

[50] M. Brandenburger, C. Cachin, M. Lorenz, and R. Kapitza, "Rollback and forking detection for trusted execution environments using lightweight collective memory," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 157–168.

[51] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "{SPORC}: Group collaboration using untrusted cloud resources," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[52] D. Mazieres and D. Shasha, "Building secure file systems out of byzantine storage," in *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, 2002, pp. 108–117.

[53] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket, "Venus: Verification for untrusted cloud storage," in *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, 2010, pp. 19–30.

[54] M. F. Madsen, M. Gaub, M. E. Kirkbro, and S. Debois, "Transforming byzantine faults using a trusted execution environment," in *2019 15th European Dependable Computing Conference (EDCC)*, 2019, pp. 63–70.

[55] S. Ghemawat and J. Dean, "Leveldb," https://github.com/google/leveldb, 2022.

[56] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[57] Apache, "Apache cassandra," https://cassandra.apache.org/_/index.html, 2022.

[58] Facebook, "Rocksdb," https://github.com/facebook/rocksdb, 2022.

[59] K. Huang, Z. Shen, Z. Jia, Z. Shao, and F. Chen, “Removing Double-Logging with passive data persistence in LSM-tree based relational databases,” in *20th USENIX Conference on File and Storage Technologies (FAST 22)*. Santa Clara, CA: USENIX Association, Feb. 2022, pp. 101–116. [Online]. Available: https://www.usenix.org/conference/fast22/presentation/huang

[60] W. Zhong, C. Chen, X. Wu, and S. Jiang, “REMIX: Efficient range query for LSM-trees,” in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 51–64. [Online]. Available: https://www.usenix.org/conference/fast21/presentation/zhong

[61] H. Chen, C. Ruan, C. Li, X. Ma, and Y. Xu, “SpanDB: A fast, Cost-Effective LSM-tree based KV store on hybrid storage,” in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 17–32. [Online]. Available: https://www.usenix.org/conference/fast21/presentation/chen-hao

[62] T. Zhang, J. Wang, X. Cheng, H. Xu, N. Yu, G. Huang, T. Zhang, D. He, F. Li, W. Cao, Z. Huang, and J. Sun, “FPGA-Accelerated compactions for LSM-based Key-Value store,” in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 225–237. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/zhang-teng

[63] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[64] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, “Paxos replicated state machines as the basis of a high-performance data store,” in *Proc. NSDI’11, USENIX Conference on Networked Systems Design and Implementation*, 2011, pp. 141–154.

[65] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, “Fault-scalable byzantine fault-tolerant services,” *SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, p. 59–74, oct 2005. [Online]. Available: https://doi.org/10.1145/1095809.1095817

[66] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, “Hq replication: A hybrid quorum protocol for byzantine fault tolerance,” in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI ’06. USA: USENIX Association, 2006, p. 177–190.

[67] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, “Zyzzyva: Speculative byzantine fault tolerance,” *ACM Trans. Comput. Syst.*, vol. 27, no. 4, jan 2010. [Online]. Available: https://doi.org/10.1145/1658357.1658358

[68] A. Mostéfaoui, M. Raynal, and M. Roy, “Time-efficient read/write register in crash-prone asynchronous message-passing systems,” *Computing*, vol. 101, no. 1, pp. 3–17, 2019.

[69] L. Lamport *et al.*, “Paxos made simple,” *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[70] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui, “Deconstructing paxos,” Tech. Rep., 2001.

[71] J. Kończak, P. T. Wojciechowski, N. Santos, T. Żurkowski, and A. Schiper, “Recovery algorithms for paxos-based state machine replication,” *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 2, pp. 623–640, 2021.

[72] D. Malkhi and M. Reiter, “Byzantine quorum systems,” in *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, ser. STOC ’97, 1997, p. 569–578.

[73] J. Martin and L. Alvisi, “A framework for dynamic Byzantine storage,” in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*. IEEE, 2004, pp. 325–334.

[74] D. K. Gifford, “Weighted voting for replicated data,” in *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, ser. SOSP ’79. New York, NY, USA: Association for Computing Machinery, 1979, p. 150–162.

[75] ——, “Weighted voting for replicated data,” in *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, ser. SOSP ’79. New York, NY, USA: Association for Computing Machinery, 1979, p. 150–162. [Online]. Available: https://doi.org/10.1145/800215.806583

[76] C. Cachin, “Asymmetric distributed trust,” in *International Conference on Distributed Computing and Networking 2021*, 2021, pp. 3–3.

[77] J. Sousa and A. Bessani, “Separating the wheat from the chaff: An empirical design for geo-replicated state machines,” in *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2015, pp. 146–155.

[78] S. Y. Cheung, M. H. Ammar, and M. Ahamad, “The grid protocol: A high performance scheme for maintaining replicated data,” *IEEE Trans. on Knowl. and Data Eng.*, vol. 4, no. 6, p. 582–592, dec 1992. [Online]. Available: https://doi.org/10.1109/69.180609

[79] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, p. 463–492, jul 1990. [Online]. Available: https://doi.org/10.1145/78969.78972

[80] M. Burke, A. Cheng, and W. Lloyd, “Gryff: Unifying consensus and shared registers,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 591–617.

[81] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann, 1996.

[82] T. D. Chandra, R. Griesemer, and J. Redstone, “Paxos made live: An engineering perspective,” in *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC ’07, 2007, p. 398–407.

[83] L. Lamport, “Paxos made simple,” *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, December 2001.

[84] R. Van Renesse and D. Altinbuken, “Paxos made moderately complex,” *ACM Comput. Surv.*, vol. 47, no. 3, feb 2015.

[85] B. Dinis, “Sloth,” https://gitlab.mpi-sws.org/bdinisa/sloth, 2022.

[86] S. Hemminger, “Network Emulation with NetEm,” 2005.

[87] C. Cloud, “Amazon elastic compute cloud,” 2009.

[88] Google, “flatbuffers,” https://github.com/google/flatbuffers, 2022.

[89] O. Project, “Openssl,” https://github.com/openssl/openssl, 2022.

[90] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil, “The log-structured merge-tree (lsm-tree),” *Acta Inf.*, vol. 33, no. 4, p. 351–385, jun 1996. [Online]. Available: https://doi.org/10.1007/s002360050048

[91] Google, “grpc,” https://github.com/grpc/grpc, 2022.

A

Correctness Proofs

In this appendix, we provide the proofs of correctness for the protocols presented in the paper. The appendix is organized as follows: A.1 presents a proof of an auxiliary theorem regarding RR quorum systems; A.2 contains a proof of correctness of the distributed register protocol; and A.3 a proof of correctness of the state machine replication protocol.

A.1 RR Quorum Systems

From the parameterization of RR quorum systems derived in Section 3.2.3, we can prove a theorem that characterizes the properties we obtain from these systems and their quorum sizes.

Theorem 2. *The quorum system defined by arbitrary subsets with the cardinalities in Equations (3.3,3.5,3.6) is a Dissemination Quorum System [72].*

Proof 3. *The D-Consistency property from Definition 5.1 in [72] follows from the fact that constraint in Equation (3.1) was preserved throughout the derivation. In particular, in the final quorum sizes we consider, we have that the read and a write quorum intersect in the following minimum number of replicas:*

$$\begin{aligned} W_Q + R_Q - N &= \\ M_R + \Delta_W + R_Q - M_R - F - \Delta_W - \Delta_N &= \\ R_Q - F - \Delta_N &= \\ F + \min(s, M_R) + \Delta_N + \Delta_R - F - \Delta_N &= \\ \min(s, M_R) + \Delta_R & \end{aligned}$$

which obeys the D-Consistency property because it contains at least one non-rolled back replica ($\Delta_R > 0$*), which directly implies that such a replica does not belong to any set* $B \in \mathcal{B}$ *according to the definition in [72].*

Similarly, the D-Availability property follows from the fact that the liveness equations were preserved throughout the derivation. In particular, in the final quorum sizes we consider, we have that:

$$\begin{aligned} N - W_Q &= \\ M_R + F + \Delta_W + \Delta_N - M_R - \Delta_W &= \\ F + \Delta_N & \end{aligned}$$

and

$$\begin{aligned} N - R_Q &= \\ M_R + F + \Delta_W + \Delta_N - F - \min(s, M_R) - \Delta_N - \Delta_R &= \\ M_R + \Delta_W - \Delta_R - \min(s, M_R) &\geq f' \end{aligned}$$

which holds given the read liveness conditions from Equation 3.7.

□

A.2 Distributed Register

In this section we prove the safety of the distributed register protocol, namely that it obeys linearizable semantics.

A.2.1 Specification

The specification is simply stated as linearizability of an object that supports read and write operations. Linearizability states that there exists a sequential history (or linearization) of the operations that took place in a history of the execution of the system, such that the linearization leads to the same outputs according to a sequential specification, and is compatible with the real time precedence of the operations. More precisely, this can be stated as:

> **R-L1**: there is a total order $<$ of operations (reads and writes), consistent with the real-time invocation/reply order (meaning that if op_1 returns before op_2 is invoked then $op_1 < op_2$)
>
> **R-L2**: reads return the value written by the most recent write according to that order.

Note that a formal proof that these properties imply linearizable semantics can be found in [81].

A.2.2 Proof

R-L1 For proving the existence of this order, we construct the the total order $<$ as the lexicographic order of timestamps $< sequencenumber, id >$. However, this is not yet a total order, since reads will have the same timestamp as the operation that created the value that was returned. We thus add the additional constraint that reads are ordered immediately after the operation that wrote the value that was read. When several read operations return the same value, then we order the reads among themselves in any total order that is compatible with the real-time order of invocation, i.e., with the constraint that if $read_1$ returns before $read_2$ is invoked then $read_1 < read_2$.

Given this construction, we now need to proof that this order is consistent with the real-time precedence order, i.e., that if operation op_2 is invoked after operation op_1 returns, then $op_1 < op_2$.

If op_1 is a read, then it only concludes after either writing back the return value to a write quorum, or reading unanimously from a write quorum, or being signaled by the stable flag that the value was previously present at a write quorum. Therefore, according to Theorem 2, a subsequent first phase of a read or write (op_2) will see that timestamp at its read quorum (or, given that timestamps grow monotonically at each replica, a higher one), and will therefore be serialized at a subsequent point in the total order either by picking a higher timestamp for writes, or by returning the same or a higher timestamp for reads. (In case op_2 is a read that returns the same timestamp, then it obeys the required property directly by the way that $<$ is constructed for that particular case.)

This argument also applies when op_1 is a write, since the second phase of each operation also reaches a write quorum.

R-L2 This follows directly from the way that the total order $<$ is constructed. In particular, reads are serialized after the write with the same timestamp, which is the write that wrote the value that is returned by the read.

A.3 State Machine Replication

In this section, we prove the safety of the state machine replication protocol, again using linearizable semantics as the notion of correctness.

A.3.1 Specification

The specification is stated as linearizability of an object that supports read and update operations. By the definition of linearizability, there exists a sequential history (linearization) of operations that took place in the execution of the system. This sequential history must be compatible with the real time order of operations and be equivalent (i.e., lead to the same outputs as) a sequential specification of the state machine. More precisely, it can be stated as:

> **SM-L1**: there is a total order $<$ of state machine operations (reads and updates), consistent with the real-time invocation/reply order (i.e., if op_1 returns before op_2 is invoked, then $op_1 < op_2$);
>
> **SM-L2**: state machine operations return the value that results from the sequential execution of the sequence of preceding operations according to that order.

As in the previous proof, the same helper theorem [81] can be used to show that these properties imply linearizable semantics, since it can apply to any atomic object, including a state machine.

A.3.2 Proof

SM-L1 For proving the existence of this order, we construct the the total order $<$ as the lexicographic order of slot numbers. However, this is not yet a total order, since reads will have the same slot number as the operation that created the value that was returned. We thus add the additional constraint that reads are ordered immediately after the operation that wrote the value that was read. When several read opearations return the same value, then we order the reads among themselves in any total order that is compatible with the real-time order of invocation, i.e., with the constraint that if $read_1$ returns before $read_2$ is invoked then $read_1 < read_2$.

Given this construction, we now need to proof that this order is consistent with the real-time precedence order, i.e., that if operation op_2 is invoked after operation op_1 returns, then $op_1 < op_2$.

If both operations are updates, this follows from the construction of the protocol, wherein the leader chooses the slot position for the update and then, by collecting a super quorum of accepts, guarantees that no subsequent update will be assigned to that position (or a lower one), and no subsequent read will see a lower slot number. Furthermore, for the case that the first operation is a read, it will require a unanimous read quorum, and additionally the replicas in this read quorum have not sent out accept messages to other replicas. Given these protocol mechanisms, the intersection property implied by Theorem 2 (read quorums intersect super quorums at a correct replica) ensure that there is no update committed to a slot number higher than the one returned in the log of any replica.

SM-L2 This follows from the way that the total order $<$ is constructed. In particular, (1) reads are serialized after the update with the same slot number, which is the latest update in the sequence of updates that leads to a state of the state machine that corresponds to the value that is returned by the operation, and (2) updates are serialized according to their slot number order, with is also the order in which replicas execute those updates, leading to outputs that are the same as the sequential specification of the state machine.