RR: A Fault Model for Efficient TEE Replication

Baltasar Dinis[123]
baltasar.dinis@tecnico.ulisboa.pt

Peter Druschel[3]
druschel@mpi-sws.org

Rodrigo Rodrigues[12]
rodrigo.miragaia.rodrigues@tecnico.ulisboa.pt

Instituto Superior Técnico[1]
INESC-ID (Lisbon)[2]
Max Planck Institute for Software Systems[3]

***Abstract*—Trusted Execution Environments (TEEs) ensure the confidentiality and integrity of computations in hardware. Subject to the TEE's threat model, the hardware shields a computation from most externally induced fault behavior except crashes. As a result, a crash-fault tolerant (CFT) replication protocol should be sufficient when replicating trusted code inside TEEs. However, TEEs do not provide efficient and general means of ensuring the freshness of external, persistent state. Therefore, CFT replication is insufficient for TEE computations with external state, as this state could be rolled back to an earlier version when a TEE restarts. Furthermore, using BFT protocols in this setting is too conservative, because these protocols are designed to tolerate arbitrary behavior, not just roll-back during a restart.**

In this paper, we propose the restart-rollback (RR) fault model for replicating TEEs, which precisely captures the possible fault behaviors of TEEs with external state. Then, we show that existing replication protocols can be easily adapted to this fault model with few changes, while retaining their original performance. We adapted two widely used crash fault tolerant protocols — the ABD [4] read/write register protocol and the Paxos [35] consensus protocol — to the RR model. Furthermore, we leverage these protocols to build a replicated metadata service called *TEEMS*, and then show that it can be used to add TEE-grade confidentiality, integrity, and freshness to untrusted cloud storage services. Our evaluation shows that our protocols perform significantly better than their BFT counterparts (between 1.25 and $55\times$ better throughput), while performing identically to the CFT versions, which do not protect against rollback attacks.

I. Introduction

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, a state machine replication (SMR) protocol like Paxos [35] may be used for this purpose.

Networked services implemented inside a trusted execution environment (TEE) like ARM TrustZone [3], AMD SEV-SNP [33], [2], Intel SGX [21], or the future Intel TDX [31] and ARM CCA [26] 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 [38], monotonic counters for ensuring state freshness [43], in-memory key-value stores [6], as well as broadcast and common random number generator primitives [32].

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 [38], [43] 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.

In this paper, we fill a gap in distributed replication research by introducing 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 tolerance (CFT) and BFT. 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

Network and Distributed System Security (NDSS) Symposium 2023
28 February - 4 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.23xxx
www.ndss-symposium.org

required communication, particularly for read operations, in the common case where restarts are infrequent.

To demonstrate the potential of the RR model, we apply it to two types of replication protocols, namely the Attiya, Bar-Noy, Dolev (ABD) read/write replication protocol and the Paxos state machine replication (SMR) protocol, showing that this adaptation is relatively straightforward. We then use these protocols to devise a replicated metadata service called *TEEMS*, which 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.

We implemented both the read/write and SMR replication protocols for three different fault models (RR, crash, and Byzantine), and we also implemented *TEEMS* and used it to implement a new class of secure cloud storage services. We evaluated the protocols using microbenchmarks and the storage system using both microbenchmarks and YCSB. Our experimental results show that the protocols for the RR model perform significantly better than their BFT counterparts, and, compared to CFT, they perform identically while additionally tolerating rollback attacks. Furthermore, *TEEMS* offers secure, sharable storage at modest overhead.

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) An extensive experimental evaluation of our protocol, and of the storage solutions based on *TEEMS* in different deployment scenarios.

The remainder of the paper is organized as follows. We motivate our work and provide some background §II, describe the RR model in §III, present the replication protocols in the new model in §IV, and an example application built using those protocols in §V. We describe our prototype implementations in §VI, evaluate them experimentally in §VII, survey related work in §VIII, and conclude in §IX.

II. Motivation: 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.

a) 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 OS, hypervisor, and platform operators.

b) 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.

c) 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 [52].

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 [52]. In particular, hardware intentionally slows the time to increment these counters to milliseconds in order to avoid wrap-around attacks [52], [43]. 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 [52]. However, trusted counters are inadequate for TEEs that frequently update their external state (e.g., a database or key-value store) 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 [9], i.e., guaranteeing that no duplicate TEE are instantiated with access to the same stored state. We discuss fork protection and how it can be achieved in replicated systems in Section VIII.

d) 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.

e) 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 (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 [38], [43]. 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 next section, 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.

III.RR MODEL

In this section, we define the RR fault model, which serves as a foundation to build replicated TEE-based systems. Then, we derive requirements for intersecting quorums, a key element to the correctness of replication protocols.

A. *RR model definition*

Nodes in the RR model are TEE instances 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, signalling that they have restarted and thus may have been subject to a rollback. The integrity properties of TEEs imply that a node 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 IV). The integrity of TEEs implies that, other than possibly holding stale persistent state after restarting, nodes correctly execute the expected code. Figure 1 illustrates the state transitions a TEE-based node can go through.

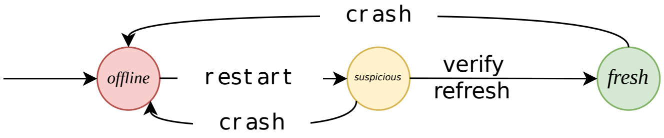


Fig. 1: States of a TEE node in the RR model. Compared to the crash-fault model, there is an additional "Suspicious" state.

B. *Background: 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 [41], [42] 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 [23]. 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 [41].

C. *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). Furthermore, 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.

D. 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.

a) 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:

$$R_Q > N - W_Q + \min(s, M_R) \qquad (1)$$

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:

$$R_Q = N - W_Q + \min(s, M_R) + \Delta_R \qquad \Delta_R > 0 \qquad (2)$$

b) 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.,

$$\begin{aligned} &W_Q > M_R \\ &W_Q = M_R + \Delta_W \qquad \Delta_W > 0 \end{aligned} \qquad (3)$$

c) 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:

$$\begin{aligned} &N - F \geq W_Q \\ &N = W_Q + F + \Delta_N \qquad \Delta_N \geq 0 \end{aligned} \qquad (4)$$

d) Final Derivation. The formulae above allow us to arrive at a precise formulation for the system and quorum sizes. In particular, by combining 3 and 4, we obtain:

$$N = M_R + F + \Delta_W + \Delta_N \qquad \Delta_W > 0, \Delta_N \geq 0 \qquad (5)$$

Then, by replacing W_Q (3) and N (5) in Equation 1, we obtain the following equation for read quorums.

$$R_Q = F + \min(s, M_R) + \Delta_N + \Delta_R \quad \Delta_R > 0, \Delta_N \geq 0 \qquad (6)$$

e) 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:

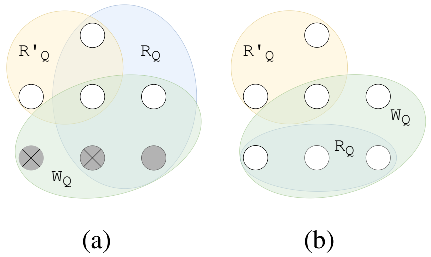
$$N - f' \geq R_Q \Leftrightarrow f' + \min(s, M_R) \leq M_R + \Delta_W - \Delta_R \qquad (7)$$

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 7:

$$\Delta_W \geq \Delta_R + F - M_R \qquad (8)$$

$$\Delta_W \geq 1 + F - M_R \qquad (9)$$



(a) (b)

Fig. 2: Example of a 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**.

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:

$$\Delta_W = \max(1, 1 + F - M_R) \tag{10}$$

$$\Delta_W = 1 + \max(F - M_R, 0) \tag{11}$$

This results in these possible deployment parameters:

$$N = \max(M_R, F) + F + 1 \tag{12}$$

$$W_Q = \max(M_R, F) + 1 \tag{13}$$

$$R_Q = F + \min(s, M_R) + 1 \tag{14}$$

f) 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 write quorum (i.e., $\max(R_Q, W_Q)$ replies). We dub these quorums *super quorums* and use S_Q to represent their size.

g) 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 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.

h) Reflections on the parametrization 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.

i) 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 [24], where a replicated object has a certain number of votes and in order to read the value, M_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 [11], 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 [30], [50], [27] or reducing the asymptotic complexity of quorums at the expense of more replicas [16].

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.

IV. 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.

A. 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 integrity guarantees of TEEs, 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 III-D. Similarly, reconfiguration is a key aspect of practical replication protocols. Quorum sizes for these reconfiguration protocols also need to be adapted accordingly.

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.

Finally, if the adapted replication protocol is to be configured such that $F < M_R$, read quorums may not intersect (see quorum property **I2** in Section III-D). Here, two disjoint read quorums may ion 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). When adapting protocols, designers must take this into account and implement mechanisms to ensure that a single value is reported by all read quorums, if required. Next, we will present two example protocols that follow these principles and illustrate some of the above challenges.

B. Read-write register

In this section, we present an adapted version of the ABD [4] protocol for a distributed read/write register with linearizable semantics[1] [29] 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 having good performance due to a simple message pattern that is linear in the number of replicas [10]. Figure 3 shows the pseudocode for the read operation while the write logic is shown in Figure 4. Since the code follows closely the original ABD protocol, our explanation highlights where we adapted the protocol to the new fault model.

```
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 $ht.stable == \texttt{true}$ 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$(ht, val(ht))$
```

Fig. 3: Pseudo-code for the register read operation.

a) 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.

b) 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.

c) 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 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. 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 for a larger 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.

d) Stabilization. To address this issue, we introduce an extra asynchronous phase, called the *stabilization* phase. This phase

[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.

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

Fig. 4: Pseudo-code for the register write operation.

```
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
```

Fig. 5: Pseudo-code for the normal case SMR operation.

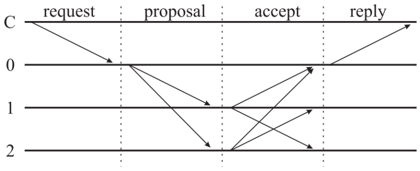


Fig. 6: Diagram of a normal case SMR operation

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 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.

e) 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 [39], 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 III 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.

C. 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 [35] 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], [15]. 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.

a) Normal case operation. We follow the multi-Paxos protocol [36], [15], 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 5, and is depicted in Figure 6 (following the protocol description from [34]). In summary, the leader replica proposes a sequence number 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.

b) 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 [14] or the Paxos description by van Renesse and Altinbuken [53]. 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.

Applying this optimization requires careful reasoning to avoid violating the linearizable semantics of the protocol. 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 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**).

c) 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 [39]. 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.

D. Security Properties

If the threat model for TEEs explained in Section II 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).

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 I: *TEEMS* Interface

V. Metadata service for trusted cloud storage

As an example application built using the protocols from Section IV, we designed and implemented a trusted metadata service for securing cloud storage, which is an important research problem by itself.

A. Motivation

TEEs have enabled services like Azure Confidential Computing [48] and Google Confidential VMs [25], 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. 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.

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 that extend the strong guarantees of today's TEE-based cloud compute services to persistent storage. The full *TEEMS* interface is summarized in Table I.

B. TEE-grade cloud storage with TEEMS

Clients can use *TEEMS* to lend untrusted cloud storage TEE-level guarantees, by using the API in Table I. *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 significantly 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.

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. 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 (one of) the untrusted storage systems. 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 $MAC(d_i)$.

While *TEEMS* ensures the confidentiality, intgrity, and freshness of stored data blobs, the untrusted storage us relied upon for availability of data blobs. For increased availability, clients may storage multiple copies of a data blob (or erasure-coded fragments) at independent storage providers. 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.

C. Leveraging different storage protocols

TEEMS implements the metadata read/write operations efficiently using our RR-tolerant ABD protocol (Section IV-B). 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 IV-C). 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.

```
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
```

Fig. 7: 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)

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 epoch number has advanced. 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 7. 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.

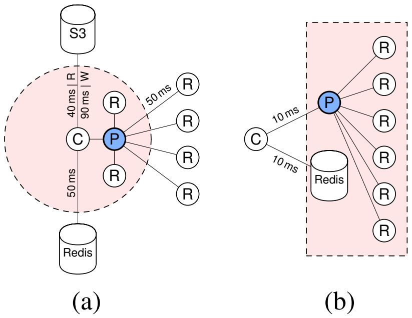


Fig. 8: Deployment scenarios: (a) cloud client, (b) colocation center. The shaded area represents a LAN (or a data center). Latencies will be used in the evaluation. C is the client, R is a replica and P is the proxy replica.

D. 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 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 datacenter, with the remaining replicas in other administrative domains (Figure 8a). 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 8b).

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.

VI. Implementation

We implemented both the read/write register and SMR protocols in the three fault models (crash, RR, and Byzantine) and the *TEEMS* prototype 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 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 3KLoC (SMR) and 6KLoC (*TEEMS*) each. Additionally, the client libraries, which interact with the replicas and, in the case of *TEEMS*, interact with the untrusted storage, take up between 2KLoC (SMR) and 5KLoC (*TEEMS*) each.

VII. Evaluation

We evaluate our various implementations based on the RR model using micro-benchmarks for protocol implementations and benchmark workloads for the full system built using those protocols. Our experiments attempt to answer the following questions:

1) How significant is the effort to change a CFT implementation to support the RR model? (§VII-A)
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? (§VII-B)
3) What is the overhead added by *TEEMS* to secure a cloud storage system under different deployments? (§VII-C)
4) What is the performance of this system when used to store metadata for a key-value store (Redis) under a benchmark workload (YSCB)? (§VII-D)

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. All machines were connected to same local network. We used `netem`[28] to emulate different deployment scenarios: 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 8; and a geo-replication scenario, based on the measured the link properties (latency and bandwidth) between several AWS EC2 [19] instances (t2.medium or t3.medium types), located in regions spread across the globe (AWS topology). This setup ensures flexibility and experimental reproducability.

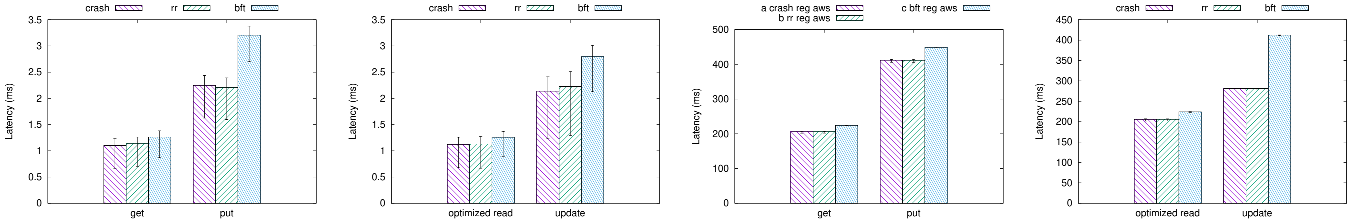
In our result graphs, each data point represents the median measurement over $3,000$ requests (except throughput numbers as described belows) and the error bars show the 5th and 95th percentiles.

A. Code changes

Changing a CFT implementation to reflect the protocol changes required by the RR model requires low programming overhead. Mainly, the adaptation needs to handle the restart flag, the existence of different quorums and the mechanisms to prevent split brain. For the read/write register prototype, we modified 72 LoC and added 211 new ones. For the SMR prototype, we modified 24 LoC and added 28.

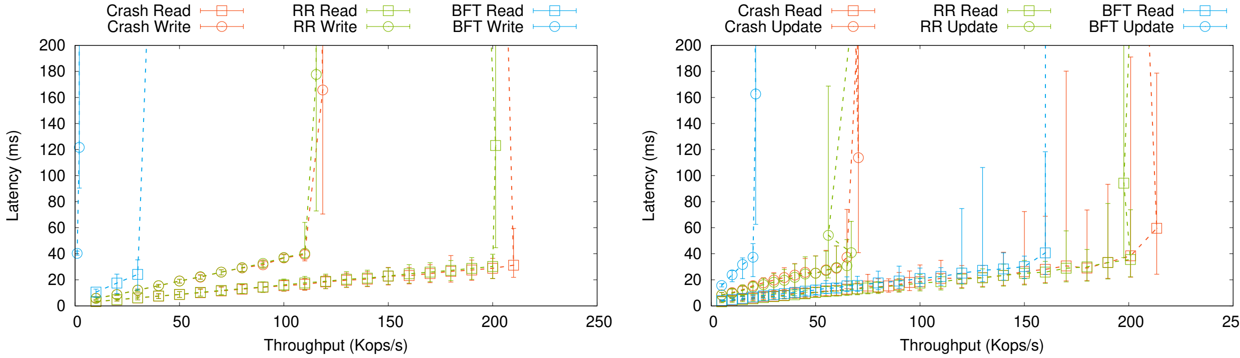
B. Protocol performance in different models

Next, we focus on comparing the performance of both classes of protocols under different fault models. We compare our adaprted read-write register with the original ABD protocol and the BFT read-write register protocol described in [41]. For SMR, we compared our protocol with Paxos as described by Kirsch and Amir [34] and the PBFT protocol [14]. We fixed the fault parameter, $F = 2$ across quorum systems (with $M_R = 2$ in the RR quorum system), yielding quorums with 3 replicas



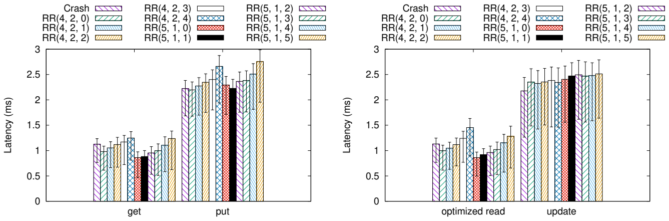
(a) Read-write register ($1ms$) (b) State machine ($1ms$) (c) Read-write register (AWS) (d) State machine (AWS)

Fig. 9: Operation latency for different protocols in different network topologies



(a) Read-write register (b) Replicated state machine

Fig. 10: Throughput-Latency curve for different protocols



(a) Read-write register (b) State machine

Fig. 11: 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.

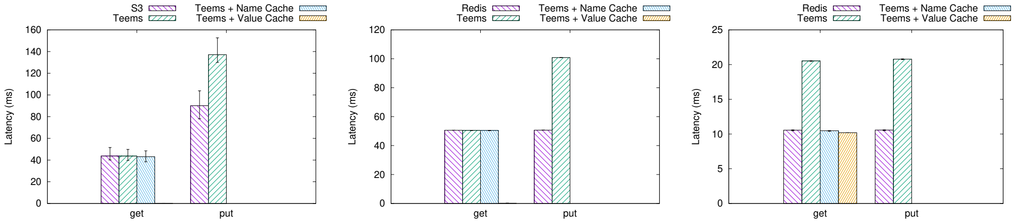
in the crash and RR models and 5 replicas in the Byzantine model.

The results in Figures 9a- 9d 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 matches that of the CFT ones, which is to be expected as they have equally sized quorums. In the particular case of the BFT read-write register, the digital signatures required by the protocol also contribute to its higher latency. Note that the overhead of the digital signatures is subsumed by the network latency cost in the AWS topology.

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. Throughput is measured by counting the 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 10a 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, respectively, a result of larger quorums and expensive digital signatures.

In Figure 10b 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.

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 III and IV, 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



(a) Client in the Cloud w/ S3 (b) Client in the Cloud w/ Redis (c) Colocation Center w/ Redis

Fig. 12: 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.

space of RR quorums and their performance difference to CFT, we reran the latency experiments from Figures 9a–9b, but with different quorum configurations. In Figures 11a–11b, 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 smaller quorums than BFT and, for read-write, the lack of digital signatures.

C. TEEMS-based storage

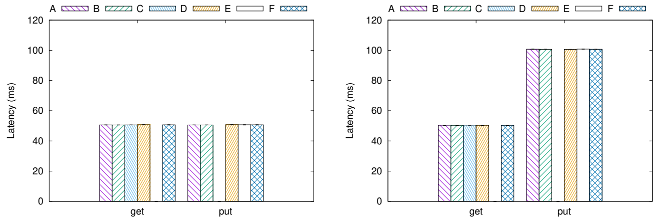
Next, we focus on understanding the performance of our example application, the *TEEMS*-based secure storage service, under different deployments, which differ in the relative location of the client, the metadata servers, and the type and location of the untrusted storage. In this set of experiments, 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 8.

a) Client in the Cloud. In this deployment (Figure 8a), the client is co-located with three metadata servers and the remaining four servers are in another datacenter. We use two variants of storage: a remote Redis deployment, and S3.

b) Collocation Center. In this scenario (Figure 8b), 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 12a–12c, 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 of both data and metadata, outperforming the baseline.



(a) Redis baseline (b) *TEEMS* + Redis

Fig. 13: Latency with YCSB workloads

D. 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 13b and 13a show that writes incur a 2× 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.

VIII. Related Work

a) Replication and fault tolerance. 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. RR is strictly more encompassing than the crash and omission models, allowing for fault behaviors that are not tolerated by these. Compared to BFT, it is tailored to the behavior of TEEs, leading to simpler, more efficient protocols with smaller quorums in the common case. Finally, compared to hybrid crash-BFT models [45], [5], [20], [18], 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.

Prior work proposed replicated systems where each replica runs in a trusted execution environment, namely to build a secure in-memory key-value store [6], a monotonic counter to protect against rollback attacks [43], payment channels in a blockchain [38], broadcast and common random number generator primitives [32], or generic transformations of crash protocols to tolerate Byzantine behavior [40]. These proposals, however, resorted to replication protocols in traditional models, which varied depending on the trust model: either they trust the correctness of the TEE and assume that TEEs cannot restart from persistently stored state, in which case they can use CFT replication [32], [6], or they pessimistically assume that adversaries can break the TEE protections, in which case they need BFT (or hybrid crash-BFT) protocols [38], [43], [40], with intrinsic additional complexity and overheads.

There are other proposals that augment the replica logic with a small trusted module [17], [37] (e.g., a trusted monotonic counter). However, their technical solutions are inherently different from the ones we adopt, because they have a small trusted computing base that is constrained by the hardware. In contrast, we focus on a setting where the entire computation is trusted but the persistent storage is not.

The RR model leads to dynamic quorum sizes, where this size depends on the number of replies from recently restarted nodes. As we discussed in Section III, prior proposals that also resort to asymmetric quorums [30], [50], [27], [11], [16] work on traditional models (CFT or BFT) and come up with protocol techniques based on different quorum sizes to improve performance or encode trust assumptions. In contrast, RR is a new model (based on a different set of assumptions) that naturally leads to dynamic and asymmetric quorums.

b) 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 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 [47]. 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 [21]. 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 [46]. 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 [21]. 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 [43] 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.

c) 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 [46] relies on a TPM [1] 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 [51] 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 [52] 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 [43] 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 [45], [18], 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.

d) 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 [43], [52], [17], [37], [46], on client cooperation/intervention [8], [12], [13], [9], [22], [44], [49] or on some centralized (possibly replicated) source of truth [7]. 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.

IX.Conclusion

This paper introduces the RR fault model, which captures precisely the fault behavior of TEEs, and applies the model to read/write and state machine replication protocols. Moreover, we designed and implemented a metadata service called *TEEMS*, and used it build a system that brings TEE-grade security as well as concurrent sharing to cloud storage. Our evaluation shows that the new fault model is more efficient than BFT and performs identically to systems tolerating crash faults, but with stronger security guarantees for replicated TEEs. It also shows that it is possible to extend existing cloud storage services to offer TEE-grade protection at a modest overhead.

References

[1] "TPM main specification level 2." Trusted Computing Group. Version 1.2, Revision 116., Mar. 2011.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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

[17] 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.

[18] 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.

[19] C. Cloud, "Amazon elastic compute cloud," 2009.

[20] 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.

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

[22] 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.

[23] 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.

[24] ——, "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

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

[26] 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.

[27] 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.

[28] S. Hemminger, "Network Emulation with NetEm," 2005.

[29] 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

[30] 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.

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

[32] 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.

[33] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," White paper at https://developer.amd.com/wordpress/media/2013/12/AMD_M

emory_Encryption_Whitepaper_v7-Public.pdf, 2016, accessed: 2020-05-24.

[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] 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/

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

[37] 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.

[38] 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

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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

[46] 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.

[47] 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.

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

APPENDIX

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

A. RR Quorum Systems

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

Theorem 1. *The quorum system defined by arbitrary subsets with the cardinalities in Equations (3,5,6) is a Dissemination Quorum System [41].*

Proof. The D-Consistency property from Definition 5.1 in [41] follows from the fact that constraint in Equation (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 [41].

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 7.

□

B. Distributed Register

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

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 [39].

2) Proof

a) 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 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 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 1, 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.

b) 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.

C. 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.

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 inkoved, 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 [39] can be used to show that these properties imply linearizable semantics, since it can apply to any atomic object, including a state machine.

2) Proof

a) 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 1 (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.

b) 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.