TEE is not a Healer: Rollback-Resistant Reliable Storage

Sadegh Keshavarzi
University of Surrey, Guildford, UK

Gregory Chockler
University of Surrey, Guildford, UK

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain

Abstract

Recent advances in secure hardware technologies, such as Intel SGX or ARM TrustZone, offer an opportunity to substantially reduce the costs of Byzantine fault-tolerance by placing the program code and state within a secure enclave known as a *Trusted Execution Environment (TEE)*. However, the protection offered by a TEE only applies during program execution. Once power is switched off, the non-volatile portion of the program state becomes vulnerable to *rollback* attacks wherein it is undetectably reverted to an older version. In this paper we consider the problem of implementing reliable read/write registers out of failure-prone replicas subject to state rollbacks. To this end, we introduce a new unified model that captures multiple failure types that can affect a TEE-based system and establish tight bounds on the fault-tolerance of register constructions in this model. We consider both the static case, where failure thresholds hold throughout the entire execution, and the dynamic case, where any number of replicas can roll back, provided these failures do not occur too often. Our dynamic register emulation algorithm, TEE-Rex, provides the first correct implementation of a distributed state recovery procedure that requires neither durable storage nor specialized hardware, such as trusted monotonic counters.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Trusted execution environments, fault tolerance, crash recovery

Digital Object Identifier 10.4230/LIPIcs.DISC.2025.18

Related Version *Extended Version*: https://arxiv.org/abs/2505.18648 [26]

Funding This work was partially supported by the projects BYZANTIUM and DECO funded by MCIN/AEI, and REDONDA funded by the CHIST-ERA network.

1 Introduction

Tolerating Byzantine failures plays a major role in designing modern reliable distributed systems, and in particular blockchains. Unfortunately, tolerating Byzantine failures is expensive in terms of message complexity [15], latency [2, 43], and the number of replicas required [16, 32, 39]. Recent advances in secure hardware technologies, such as Intel SGX or ARM TrustZone, offer an opportunity to substantially reduce these costs by placing the program's code and a portion of its state in a secure enclave known as a *Trusted Execution Environments (TEE)*. Applications can furthermore detect any runtime alterations to the state stored outside the enclave using integrity metadata stored inside it. These mechanisms together can then be used to convert Byzantine failures to simple crash or omission failures, which are less expensive to tolerate [7, 31, 41, 42].

However, a TEE is a not a panacea, because the protection it offers only applies during program execution. Once power is switched off, the non-volatile portion of the program state becomes vulnerable to tampering by an attacker. To address this, hardware manufacturers

39th International Symposium on Distributed Computing (DISC 2025).
Editor: Dariusz R. Kowalski; Article No. 18; pp. 18:1–18:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

provide a *sealing* mechanism [12] that allows programs to encrypt a portion of their state using a device-specific key before storing it in non-volatile storage. When the machine restarts, the TEE can check whether the sealed state was indeed written by it in the past. However, it cannot determine whether this state is the most recent version. This limitation makes TEE-based systems vulnerable to *rollback* attacks where the sealed state is undetectably replaced with an older version [8, 13, 35, 40, 42].

In this paper we study theoretical foundations for building reliable distributed systems in the presence of rollbacks. We start by introducing *crash-restart-rollback* (CRR), a new unified failure model that formalizes rollbacks alongside other types of failures that can occur in a TEE-based system. In this model, processes can use local non-volatile storage to persist their states and are assumed to behave correctly while they are executing. A CRR-faulty process can permanently crash, crash and restart infinitely often, or crash and restart with the content of its non-volatile storage being undetectably reverted to an older version.

We then study the problem of implementing reliable read/write registers in an asynchronous message-passing system where the register's state is stored at a collection of n CRR failure-prone *replicas* accessed by arbitrarily many crash-prone *clients*. We consider two sub-classes of the above CRR failure model for replicas – *static* and *dynamic*. Static failure models assume the existence of a priori fixed thresholds restricting the number of replicas that can experience failures of various types in every execution; this is similar to the classical crash or Byzantine failure models. The dynamic failure models generalize the static ones by permitting any number of replicas to roll back in exchange for requiring that these failures do not occur too often.

Static failure models. For the static failure models, we give a fine-grained characterization of failure resilience, time complexity, and non-volatile storage requirements of a register implementation in terms of several thresholds: k on the total number of any CRR failures; $r \leq k$ on the number of rollback failures; and $b \leq n$ on the number of correct replicas that can crash at least once, but eventually stay up. Our main result establishes the following: a wait-free atomic multi-writer/multi-reader (MWMR) register can be implemented in the static model if and only if $n \geq 2k + \min(b, r) + 1$. Our algorithm matching this bound has the latency of 4 message delays for both writes and reads, which is the same as for crash fault-tolerant register implementations [11, 33] and is thus optimal. This stands in a sharp contrast with the Byzantine-resilient register constructions, where write and read latencies of 4 and 8 message delays are inherent even for implementing a single-writer/multi-reader (SWMR) wait-free atomic register [14].

Our lower bound implies that, when $r \leq b$, rollback failures are no different from Byzantine failures in terms of failure resilience: the number of replicas $n \geq 2k + r + 1$ required in this case is the same as that needed to implement Byzantine-resilient registers when at most r out of k faulty replicas can behave arbitrarily [18]. On the other hand, the two models are separated when $b < r$: in this case, a register can be implemented in CRR with $2k+b+1$ replicas, which is strictly fewer than $2k+r+1$ required by Byzantine fault-tolerant implementations.

We also establish a lower bound on non-volatile storage consumption: when $2k + r + 1 \leq n < 2k + b + 1$, at least $2k + r + 1$ replicas must store their states persistently on their local non-volatile storage. Since this storage is not protected by TEEs, in practice applications must rely on expensive mechanisms to monitor its integrity at runtime, e.g., using TEE-protected integrity metadata, such as Merkle trees. Our results show that for some resilience levels these overheads are unavoidable.

Dynamic failure models. For the dynamic failure models, we first show that no implementation of a single-writer/single-reader (SWSR) safe register can use fewer than $d + c + 1$ replicas, where d bounds the number of replicas that either crash permanently or never stop crashing and recovering, and $c \geq d$ bounds the number of replicas that crash at least once. While it is well-known that no register implementation can exist without a majority of replicas being eventually up ($n \geq 2d + 1$), our result further refines this lower bound for the case when $c > d$.

For a matching upper bound, we propose an algorithm, called TEE-Rex, that implements an MWMR atomic register using $n \geq d + c + 1$ replicas. The algorithm is always safe and is wait-free in executions where: $n - d$ replicas eventually stop crashing; and either replicas do not crash too often or $n - c$ replicas never crash at all. Unlike in the static model, these conditions place no restrictions on the number of replicas that can roll back. The algorithm is parameterized by d, but is unaware of c. It achieves the optimal time complexity of 4 message delays for both writes and reads in failure-free executions.

The most interesting ingredient of TEE-Rex is a novel distributed recovery protocol that enables crashed replicas to rebuild their state and become fully operational upon restart. As we explain in §4.1, implementing recovery correctly is nontrivial: a naive approach of simply querying a read quorum and adopting the state of the most up-to-date replica is fundamentally unsafe [36]. In fact, as we discovered in this work, some of the prior recoverable register implementations suffer from nontrivial safety bugs [13, 17] (§5). Others do not guarantee liveness [35, 42], or require trusted hardware-based primitives such as persistent monotonic counters [25, 36]. In contrast, our TEE-Rex algorithm provides what we believe is the first self-contained and rollback-resilient construction of a read/write register under dynamic failure models.

2 System Model

We consider an asynchronous message-passing system consisting of a collection of failure-prone processes $\mathcal{P} = \{p_1, p_2, \ldots\}$ implementing a high-level object abstraction. The set of processes is partitioned into a set $\mathcal{R} = \{p_1, \ldots, p_n\}$ of $n > 1$ *replicas*, and a set $\{p_{n+1}, p_{n+2}, \ldots\}$ of possibly infinitely many *clients*. Similarly to the classical faulty shared memory model of [3, 24], the replicas are responsible for storing the object state, and the clients interact with the replicas to handle the requests supported by the implemented object type. To study tradeoffs between failure resilience and non-volatile storage consumption, we assume that $s \geq 0$ replicas have access to non-volatile storage.

Formally, an object *implementation* is a composition of client and replica automata. An execution of an implementation is a sequence of states interleaved with atomic send and receive actions starting from an initial state. The action atomicity ensures that all non-volatile storage modifications performed as part of the received message handling are *failure atomic* [9], i.e., their effects are either persisted in their entirety or not at all.

Failure models. A failure model is a predicate over executions. An execution α is *valid under a failure model* $\mathcal{F}$ (or simply $\mathcal{F}$*-valid*) if $\mathcal{F}(\alpha)$ holds. For failure models $\mathcal{F}$ and $\mathcal{F}'$, $\mathcal{F} \preceq \mathcal{F}'$ iff $\forall \alpha. \mathcal{F}(\alpha) \implies \mathcal{F}'(\alpha)$.

Client failures. Clients can experience permanent *crash* failures, but otherwise do not deviate from their prescribed protocols. A client is *correct* in an execution if it never crashes, and is *faulty*, otherwise.

Replica failures. Our baseline failure model for replicas, to which we refer as *crash-restart-rollback* (CRR), formalizes the types of failures that can occur in a TEE-based system (see §1). Under CRR, a replica can experience a *crash* that interrupts its execution either permanently or temporarily. In the former case, the replica stops taking steps; in the latter case, it *restarts*, by executing a specified **on restart** block and then continues its execution from the state reached thereupon. For simplicity, we assume that when a replica is restarted, the restart and its immediately preceding crash occur *simultaneously* as a single atomic event. We refer to the restarted replicas that have completed their **on restart** blocks as *active.* Upon restart, a replica with non-volatile storage may additionally experience a *rollback* failure, which causes the content of its non-volatile storage to revert to a version older than it had before the latest crash.

A replica is *up* in an interval $[t, t']$ if it does not crash without restart before t, and does not crash (either with or without restart) at all times in $[t, t']$; a replica is *up after* t, if it is up in $[t, \infty)$; a replica p_i is *eventually up* if there exists t such that p_i is up after t. Replicas in CRR fall into four disjoint classes:

- a replica is *perfect* if it never crashes;
- a replica is *benign* if it crashes and restarts at least once, but is eventually up, and furthermore, it never rolls back;
- a replica is *crash-faulty* if it crashes without restarting or crashes and restarts infinitely often, and furthermore, it never rolls back; and
- a replica is *rollback-faulty* if it rolls back at least once.

Perfect and benign replicas are together called *correct*; crash-faulty and rollback-faulty replicas are together called *faulty.*

Note that CRR generalizes the classical crash-recovery failure model [22] by extending its set of faulty behaviors with rollback failures. It is also strictly weaker than the Byzantine failure model since faulty replicas are required to follow their prescribed protocols. The relationship of our model to other prior failure models is further discussed in §5.

Channel reliability assumptions. We assume that every pair of processes p_i and p_j are connected via point-to-point authenticated links such that if both p_i and p_j are up after some t, then every message sent by p_i to p_j after t is eventually delivered by p_j. Note that this assumption is strictly weaker than the standard notion of a reliable channel. Specifically, it does not require reliability for messages sent by a correct process to another correct process before both processes stop crashing.

Specifications. We use standard safety and liveness notions to specify correctness conditions for register implementations: atomicity [28] (linearizability [21]), safeness [28], wait-freedom [19], and obstruction-freedom [20]. We consider both *single-writer/single-reader (SWSR)* and *multiple-writer/multiple-reader (MWMR)* register implementations.

Fault-tolerant implementations. An object implementation is *safe under a failure model* $\mathcal{F}$ (or simply *$\mathcal{F}$-safe*) if it satisfies safety in all $\mathcal{F}$-valid executions. We define *$\mathcal{F}$-liveness* similarly. An implementation is *$\mathcal{F}$-tolerant* if it is both safe and live under $\mathcal{F}$.

We study the implementability of registers under failure models where, in every execution, arbitrarily many clients can crash and replicas are subject to a restricted variant of the baseline CRR model above. Specifically, in §3 we consider *static* failure models $\mathsf{CRR}(k, r, b)$ for replicas where the number of faulty, rollback-faulty, and benign replicas in every execution

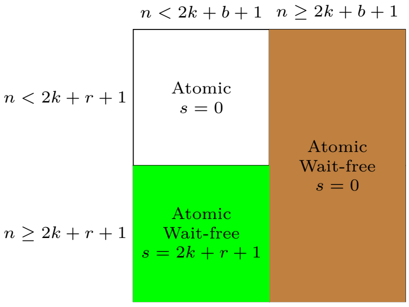

Figure 1 Resilience (k, r, b) and non-volatile storage usage (s) of the upper bound. Each square corresponds to the chunk of the problem space where the conjunction of the conditions on the axes holds.

is bounded by $k \leq n$, $r \leq k$, and $b \leq n$, respectively. Note that $\forall k, r, b.\, \mathsf{CRR}(k, r, b) \preceq \mathsf{CRR}$. In §4 we consider more flexible *dynamic* failure models.

3 Register Implementations in Static Failure Models

In this section we give a full characterization of the costs of implementing a register under static failure models in terms of its failure resilience, time complexity, and non-volatile storage requirements. For resilience, the following theorem establishes a tight bound on the total number of replicas n as a function of thresholds on the number of replicas of different types.

▶ **Theorem 1.** *Assume that all replicas have access to non-volatile storage. Then for all k, r, b, there exists a* $\mathsf{CRR}(k, r, b)$*-tolerant implementation of a wait-free atomic MWMR register if and only if* $n \geq 2k + \min(b, r) + 1$.

When $r = 0$, the bound in the theorem specializes to $n \geq 2k + 1$, the same as for crash fault-tolerant register implementations. When $r \leq b$, the bound specializes to $n \geq 2k + r + 1$. This matches the known generalization [18] of the $n \geq 3k + 1$ Byzantine implementability bound for registers [14, 34] to the case when at most r out of k faulty replicas can behave arbitrarily and the rest can crash but not deviate from the prescribed protocol. On the other hand, if $b < r$, then the theorem shows that we can implement a $\mathsf{CRR}(k, r, b)$-tolerant wait-free atomic register with $n = 2k + b + 1 < 2k + r + 1$ replicas – strictly fewer than in the Byzantine case. We next refine Theorem 1 to additionally give a tight bound on the number of replicas that must be equipped with non-volatile storage.

▶ **Theorem 2.** *Let s be the number of replicas with non-volatile storage. Then for all k, r, b, there exists a* $\mathsf{CRR}(k, r, b)$*-tolerant implementation of a wait-free atomic MWMR register if and only if*

$$(2k + r + 1 \leq n < 2k + b + 1 \wedge s \geq 2k + r + 1) \vee (n \geq 2k + b + 1).$$

Note that the constraint on n in Theorem 2 implies that in Theorem 1; we illustrate the constraint in Figure 1. The above theorem shows that, when $n \geq 2k + b + 1$, atomic registers can be implemented without any stable storage at all. In practice, avoiding non-volatile storage offers substantial performance gains when the register state fits within the TEE-protected RAM, as it avoids the overheads of monitoring storage integrity at runtime.

Finally, the result in Theorem 2 can be strengthened along several dimensions:

- The time complexity of the upper bound we present in §3.1 is always 4 message delays for both writes and reads. This matches the time complexity of crash fault-tolerant implementations, which is optimal. This property stands in a sharp contrast with Byzantine-resilient register constructions, where write and read latencies of 4 and 8 message delays, respectively, are inherent even for single-writer/multi-reader (SWMR) wait-free atomic registers [14].
- The lower bound in Theorem 2 is established for obstruction-free safe implementations of SWSR registers.
- When either of b or r is unknown, a tight bound is obtained from Theorem 2 by letting $b \triangleq n$ or $r \triangleq k$.
- If the number of replicas is less than the resilience bound ($n < 2k + \min(b, r) + 1$), the upper bound does not sacrifice safety, but only liveness (the top-left box in Figure 1). In §4 we leverage this property to develop a register construction under dynamic failure models that do not require all failure thresholds to hold for the entire execution.

Our findings collectively demonstrate that, while existing Byzantine fault-tolerant register constructions [1,14,18,34] can be used for implementing $\mathsf{CRR}(k, r, b)$-tolerant atomic registers, they are not well-suited for this purpose due to their sub-optimal failure resilience and latency. We next present an algorithm that gives a witness for our upper bound; we defer the proofs of the other results to [26, §A].

3.1 Upper Bound

An algorithm $\mathcal{A}$ that shows our upper bound for $\mathsf{CRR}(k, r, b)$ is presented in Figure 2. It is based on the MWMR variant of ABD [5,11,33], where read and write quorums are sets of replicas of size q_r and q_w, respectively. The algorithm uses the truth value of the predicate $P(n, k, r, b) = (2k + r + 1 \leq n < 2k + b + 1)$ to determine the values of q_r and q_w (line 2), the restart handler logic (lines 6–7), and whether non-volatile storage should be used to store the replica's state (line 5). In the following we refer to the variant of $\mathcal{A}$ executed if P holds as algorithm $\mathcal{A}_1$, and as algorithm $\mathcal{A}_2$ otherwise. The former corresponds to the bottom-left box in Figure 1, while the latter corresponds to the remaining portions of the figure.

Replica states. Each replica stores a copy of the register state in a state variable (line 3), which is a tuple consisting of the register value state.val and timestamp state.ts. Timestamps are pairs of a counter and the client identifier, ordered lexicographically: $(cnt_1, j_1) \leq (cnt_2, j_2)$ iff $cnt_1 < cnt_2 \vee (cnt_1 = cnt_2 \wedge j_1 \leq j_2)$. Each replica also maintains a Boolean flag stale: if this flag is true, the replica is not allowed to respond to read requests.

Quorum-access functions. The clients use auxiliary functions `read_quorum` (lines 14–19) and `write_quorum` (lines 29–34) to respectively query and update various components of replica states at a quorum. For `read_quorum`, the *req* argument specifies the state elements to query: e.g., TSVal corresponds to (state.ts, state.val). For `write_quorum`, *req* specifies which state elements to modify and their new values: e.g., TSVal(ts, v) to set (state.ts, state.val) to (ts, v).

The `read_quorum` (respectively, `write_quorum`) function starts by generating a globally unique request identifier *id* (lines 15 and 30); in practice this can be implemented using cryptographic nonces. The function then broadcasts a `READ`(id, req) (respectively, `WRITE`(id, req)) message to all replicas, and awaits `READ_ACK` (respectively, `WRITE_ACK`) messages from q_r (respectively, q_w) replicas. Since channels may fail to deliver messages sent

```
   Predicates and constants:
1      $P(n,k,r,b) \triangleq 2k+r+1 \le n < 2k+b+1$
2      $(q_w, q_r) \triangleq (n-k, n-k)$ if $P(n,k,r,b)$; and $(n-k, k+1)$ otherwise

   State:
3      state, initially $((0,0), v_0)$, with selectors ts and val
4      stale $\in$ {TRUE, FALSE}, initially FALSE
5      state, stale: non-volatile if $P(n,k,r,b) \wedge (i \le 2k+r+1)$

6  on restart
7      if $\neg P(n,k,r,b)$ then stale $\leftarrow$ TRUE

8  function write($v$)
9      $S \leftarrow$ read_quorum(TS)
10     $cnt \leftarrow \max\{cnt' \mid (cnt', \_) \in S\}$
11     $ts \leftarrow (cnt+1, i)$
12     write_quorum(TSVal($ts, v$))
13     return ack

14 function write_quorum($req$)
15     $id \leftarrow$ get_unique_id()
16     do
17         send WRITE($id, req$) to $\mathcal{R}$
18     periodically until received
         {WRITE_ACK($id, j$) | $p_j \in Q$}
         for some $Q$ such that $|Q| \ge q_w$
19     return

20 when received WRITE($id$, TSVal($ts, v$)) from $p_j$
21     if $ts >$ state.ts then
22         (state.ts, state.val) $\leftarrow (ts, v)$
23     send WRITE_ACK($id, i$) to $p_j$

24 function read()
25     $S \leftarrow$ read_quorum(TSVal)
26     let $(ts, v)$ be such that
         $(ts, v) \in S \wedge$
         $ts = \max\{ts' \mid (ts', \_) \in S\}$
27     write_quorum(TSVal($ts, v$))
28     return $v$

29 function read_quorum($req$)
30     $id \leftarrow$ get_unique_id()
31     do
32         send READ($id, req$) to $\mathcal{R}$
33     periodically until received
         {READ_ACK($id, x_j, j$) | $p_j \in Q$}
         for some $Q$ such that $|Q| \ge q_r$
34     return $\{x_j \mid p_j \in Q\}$

35 when received READ($id, req$) from $p_j$
36     pre: stale = FALSE
37     $r \leftarrow$ case $req$ do
38             TS: state.ts
39             TSVal: (state.ts, state.val)
40     send READ_ACK($id, r, i$) to $p_j$
```

Figure 2 Pseudocode for algorithm $\mathcal{A}$ at process $p_i \in \mathcal{P}$.

before their respective destinations stop crashing (§2), to ensure liveness, both `read_quorum` and `write_quorum` continue retransmitting requests until receiving the desired quorums of responses. The `read_quorum` function then returns the set of payloads received in `READ_ACK` messages from a read quorum of replicas.

Whenever a replica with stale = FALSE receives a `READ`(id, req) message (line 35), it responds with a `READ_ACK` message, carrying id and the current value of the state variable associated with req. Whenever a replica receives a `WRITE`(id, req) message (line 20), it applies the update encoded in req to its state and acknowledges this fact with a `WRITE_ACK` message. In particular, if $req =$ TSVal(ts, v) and state.ts $< ts$, then (state.ts, state.val) is set to (ts, v); otherwise, the state is left unchanged.

Read and write protocols. To write a value v, a process p_i first calls `read_quorum` to retrieve a set of timestamps from a read quorum (line 9). It then selects the highest timestamp counter cnt and calls `write_quorum` to store v with timestamp $(cnt+1, i)$ at a write quorum. To read a value, a process p_i first calls `read_quorum` to retrieve a set of timestamp-value pairs from a read quorum (line 25). It then selects the pair (ts, v) with the highest timestamp, calls `write_quorum` to store it at a write quorum, and returns v.

Algorithm $\mathcal{A}_1$. Algorithm $\mathcal{A}_1$ handles the case when the predicate $P(n, k, r, b)$ holds, i.e., $2k + r + 1 \leq n < 2k + b + 1$ (the bottom-left box in Figure 1 and the first disjunct in Theorem 2). The algorithm uses read and write quorums of size $q_w = q_r = n - k$ (line 2) and keeps the state of $2k + r + 1$ replicas in non-volatile storage (line 5), which is the minimum required by the lower bound of Theorem 2. Its restart handler returns immediately (lines 6–7), so the stale flag is always FALSE at all replicas, which can thus respond to `READ` messages.

Since in every execution at least $n - k > 0$ replicas are eventually up, both write and read quorums are always available, and, as a result, algorithm $\mathcal{A}_1$ is wait-free. We defer the full proof of its linearizability to [26, §A] and only prove the following key property the proof relies on:

▶ **Property 3** (Real-Time Order). *Let WQ be a completed call to* `write_quorum`(TSVal(x, _)), *and RQ be a completed call to* `read_quorum`(TS) *or* `read_quorum`(TSVal). *If RQ is invoked after the completion of WQ, then it returns a set containing a value $\geq x$ for* state.ts.

Proof of Property 3 for $\mathcal{A}_1$. Since WQ returns, it has executed the TSVal(x, _) request at $q_w = n - k$ replicas. Since RQ returns, it has retrieved the value of state.ts from a $q_r = k + 1$ replicas. The intersection of the two quorums has a cardinality $\geq (n-k)+(n-k)-n = n-2k$. Furthermore, the number of replicas in this intersection that store their states on non-volatile storage is $\geq (n - 2k) + (2k + r + 1) - n = r + 1$. Since at most r replicas experience rollback failures, there exists at least one replica p in the intersection of the quorums used by WQ and RQ that stores its state on non-volatile storage and never rolls back. Our protocol only allows replicas to increase the value of state.ts during normal execution (line 22), so replica p must have state.ts $\geq x$ after it responds to WQ. Since RQ starts after WQ completes, p must respond to RQ after WQ completes. Thus, p must have state.ts $\geq x$ when it responds to RQ, so RQ receives a response containing state.ts $\geq x$. ◀

Algorithm $\mathcal{A}_2$. Algorithm $\mathcal{A}_2$ handles the case when the predicate $P(n, k, r, b)$ does not hold (the right and top-left boxes in Figure 1 and the second disjunct in Theorem 2). The algorithm does not use non-volatile storage, and has write quorums of size $q_w = n - k$ and read quorums of size $q_r = k + 1$. Additionally, only replicas that have not crashed before being queried can participate in read quorums. To ensure this, each replica sets its stale flag to TRUE before returning from the recovery procedure (line 7). Recall that a replica contacted by `read_quorum` checks this flag and responds only if it is FALSE (line 36).

When $n \geq 2k + b + 1$, in every execution at least $n - k > 0$ replicas are eventually up, so some write quorum is eventually available. Also, at least $n - (k + b) \geq k + 1$ replicas never crash, so some read quorum is always available. Hence, in this case $\mathcal{A}_2$ is wait-free. Furthermore, $\mathcal{A}_2$ is always safe, even when $n < 2k + r + 1$. Similarly to $\mathcal{A}_1$, the safety of $\mathcal{A}_2$ this follows from the Real-Time Order Property, proved below; the rest of the proof is given in [26, §A].

Proof of Property 3 for $\mathcal{A}_2$. The intersection of any read and write quorums has a cardinality $\geq (n - k) + (k + 1) - n \geq 1$. Hence there exists a replica p that both executes TSVal(x, _) and responds to RQ. Since p is a member of a read quorum, it could not have crashed. Since the value of state.ts is non-decreasing in the absence of crashes (line 22), p must have state.ts $\geq x$ after it responds to WQ. Since RQ starts after WQ completes, p must respond to RQ after WQ completes. Thus, p must have state.ts $\geq x$ when it responds to RQ, so RQ receives a response containing state.ts $\geq x$. ◀

4 Register Implementations in Dynamic Failure Models

In this section we study the implementability of registers under failure models where any number of replicas can experience CRR failures throughout an execution. By Theorem 1, no register implementation can be simultaneously safe and live in all executions if $n < 2k + \min(b, r) + 1$. To circumvent this impossibility, we introduce a family of *dynamic* failure models CRR-D, defined as follows.

We assume a constant Δ such that, in any execution, every message sent by a process p_i to a process p_j at time t is guaranteed to be received by p_j by $t + \Delta$, provided p_i and p_j are up in $[t, t + \Delta]$. Note that this assumption does not make the model stronger than asynchronous: since we do not assume a lower bound on message delays or processing times, processes do not have a means to measure time passage and thus take advantage of the existence of Δ [6].

▶ **Definition 4.** *Given d, c, and M, an execution α is* CRR-D(d, c, M)*-**valid** if there exists $U \subseteq \mathcal{R}$ such that $|U| \geq n - d$, all replicas in U are eventually up in α and either:*

1. *$c > d$ and for any two crash events occurring at times t and $t' \geq t$, we have $t' - t > M\Delta$; or*
2. *no replica crashes in some set $S \subseteq U$ such that $|S| \geq n - c$.*

Intuitively, the parameter d captures the upper bound on the number of replicas that are not eventually up in α, and must be known to any register implementation. Conditions 1 and 2 further restrict failure scenarios: either failures must be separated by at least M message delays, or at most $c \geq d$ replicas can crash in α. The former is similar to the churn-limiting assumptions used to model process participation in dynamic and reconfigurable systems [6]. The parameter M is implementation-specific and captures the minimum time required for a replica to recover its state before the next crash occurs.

We show that every static failure model CRR(k, r, b) from §3 is a special case of a dynamic model CRR-D$(k, k + b, _)$. Thus, any CRR-D$(k, k + b, _)$-tolerant register implementation is also CRR(k, r, b)-tolerant.

▶ **Proposition 5.** $\forall k, r, b, M.\,$CRR$(k, r, b) \preceq$ CRR-D$(k, k + b, M)$.

Proof. Consider any CRR(k, r, b)-valid execution α. Then in this execution up to k replicas are faulty and up to b replicas are benign. Let U be the set of correct replicas, and $S \subseteq U$ be the set of perfect replicas. Then $|U| \geq n - k$ and $|S| \geq n - k - b$. Furthermore, the replicas in U are eventually up and those in S never crash, so the condition in case 2 of Definition 4 is satisfied. Thus, α is CRR-D$(k, k + b, M)$-valid, as needed. ◀

We prove the following upper bound in the dynamic failure model:

▶ **Theorem 6.** *For all d, c, if $n \geq d + c + 1$ and $M \geq 12$, then there exists an implementation of an atomic wait-free MWMR register that is* CRR-D(d, c, M)*-live and always safe.*

For the lower bound, we prove a stronger result that holds for CRR-D(d, c, M)-tolerant implementations, and not just those that are CRR-D(d, c, M)-live and always safe:

▶ **Theorem 7.** *For all d, c, M, if there exists a* CRR-D(d, c, M)*-tolerant implementation of a safe obstruction-free SWSR register, then $n \geq d + c + 1$.*

The two theorems imply a tight resilience bound:

▶ **Theorem 8.** *For all d, c,* CRR-D(d, c, M)*-live implementation of an atomic wait-free MWMR register exists for some M if and only if $n \geq d + c + 1$.*

The lower bound in Theorem 7 reveals an inherent trade-off: while by Proposition 5 any $\mathsf{CRR\text{-}D}(k, k+b, \_)$-tolerant register implementation is also $\mathsf{CRR}(k, r, b)$-tolerant, any such implementation requires $n \geq 2k+b+1$, thus sacrificing the optimal resilience under $\mathsf{CRR}(k, r, b)$ (cf. Theorem 2).

In the rest of this section we present an algorithm that validates Theorem 6, which we call TEE-Rex; we defer the proofs of the other results to [26, §B]. The algorithm's latency in crash-free executions is 4 message delays for both reads and writes, which is optimal. Since under $\mathsf{CRR\text{-}D}(d, c, M)$, any number of replicas can suffer rollbacks in some executions, the algorithm does not rely on non-volatile storage. It also does not require the knowledge of c.

4.1 Crash-Consistency Basics

The pseudocode of TEE-Rex appears in Figure 3. It reuses the `read` and `write` procedures of algorithm $\mathcal{A}$ (Figure 2, lines 8–28). The algorithm relies on read quorums of size $q_r = d+1$, and so-called *crash-consistent* write quorums of size $q_w = n-d$, introduced in the following. Given the mapping between static and dynamic models established by Proposition 5, these quorum sizes mirror those in $\mathcal{A}_2$: $k+1$ for read quorums and $n-k$ for write quorums.

Since replicas lose their memory contents during a restart, TEE-Rex follows the approach of $\mathcal{A}_2$: a replica relies on its stale flag to determine whether its current state can be used to respond to `READ` requests. However, in contrast to $\mathcal{A}_2$, simply setting $\mathsf{stale} = \text{TRUE}$ upon restart so that any replica that has not crashed permanently is active will not be live. To see why, fix arbitrary M and Δ, and consider an execution α where all replicas are eventually up, every replica crashes and restarts at least once, and every two consecutive crashes are separated by $M\Delta$. Clearly, α is $\mathsf{CRR\text{-}D}(0, n, M)$-valid. However, $\mathcal{A}_2$ will not be live in α as it eventually runs out of non-stale replicas to form read quorums. To deal with such scenarios, a key ingredient of TEE-Rex is a novel recovery protocol executed by a replica upon restart (lines 4–18). This protocol reconstructs the state of a restarted replica, and thus enables it to clear its stale flag.

The protocol achieves this by synchronizing with other replicas. However, doing this naively by simply querying a read quorum and adopting the state of the most up-to-date replica would lead to a safety violation. For example, consider the algorithm $\mathcal{A}_2$ instantiated with $q_r = d+1$ and $q_w = n-d$. Suppose we modify the **on restart** procedure at line 6 as follows. After setting $\mathsf{stale} = \text{TRUE}$, a restarting replica would invoke `read_quorum`(TSVal) to retrieve the register states from a read quorum of replicas, assign the one with the highest timestamp to its own copy of state, set $\mathsf{stale} = \text{FALSE}$, and then return. The following scenario demonstrates that this modification results in a linearizability violation.

▶ **Example 9.** Let $n = 3$, $d = 1$ and $c = 1$, so that $n \geq d+c+1$. Consider the execution of the modified algorithm in Figure 4, which is valid under $\mathsf{CRR\text{-}D}(1, 1, \_)$. First, a client p_4 invokes `write`(v) (line 8 in Figure 2), which calls `read_quorum`(TS) to query timestamps at a read quorum and then `write_quorum`($\mathsf{TSVal}((1,4), v)$) to store the new register state at a write quorum. In the latter call, p_4 sends $m = $ `WRITE`($\mathsf{TSVal}((1,4), v)$) to all replicas and waits until it gathers `WRITE_ACK` responses from a write quorum. Replica p_3 gets m first, sets $\mathsf{state} = ((1,4), v)$, and responds with a `WRITE_ACK`. It then crashes, loses its state, and reconstructs the state by contacting a read quorum $\{p_1, p_2\}$ of size $q_r = d+1 = 2$. Since m has not yet reached either replica, p_3 sets state back to its initial value $((0,0), v_0)$. Replica p_2 then receives m, sets $\mathsf{state} = ((1,4), v)$, and replies to p_4 with a `WRITE_ACK`. Process p_4 now gathers `WRITE_ACK` responses from $q_w = n-d = 2$ processes, which constitutes a write quorum. Thus, `write_quorum` and `write`(v) at p_4 terminate. Later, a client p_5 calls `read`,

```
Constants:
    $(q_w, q_r) \triangleq (n - d, d + 1)$

State:
    state, initially $((0,0), v_0, [0..0], [0..0])$, with selectors ts, val, cv, and pre_cv
    stale $\in$ {TRUE, FALSE}, initially FALSE

on restart
    stale $\leftarrow$ TRUE
    $S \leftarrow$ read_quorum(preCV($i$))
    $next\_inc \leftarrow \max(S) + 1$
    write_quorum(preCV($i, next\_inc$))
    state.cv[$i$] $\leftarrow next\_inc$
    write_quorum(CV($i$, state.cv[$i$]))
    $S \leftarrow$ read_quorum(State(state.cv[$i$]))
    forall $z = 1..n$ do
        state.pre_cv[$z$] $\leftarrow \max\{$state.pre_cv[$z$],
          $\max_{s \in S}\{s$.pre_cv[$z$]$\}\}$
        state.cv[$z$] $\leftarrow \max\{$state.cv[$z$],
          $\max_{s \in S}\{s$.cv[$z$]$\}\}$
    let $(ts, v)$ be such that $\exists s \in S$.
      $(ts, v) = (s$.ts$, s$.val$) \wedge ts = \max\{s$.ts $\mid s \in S\}$
    if $ts >$ state.ts then
        (state.ts, state.val) $\leftarrow (ts, v)$
    stale $\leftarrow$ FALSE

function write_quorum($req$)
    $(id, Q, cv) \leftarrow$ (get_unique_id(), $\emptyset$, $[0..0]$)
    do
        do
            foreach $p_z \in \mathcal{R} \setminus Q$ do
                send WRITE($id, req, cv[z]$) to $p_z$
        periodically until received
          {WRITE_ACK($id, inc_j, cv_j, j$) $\mid p_j \in K$}
          for some $K$ such that $|K \cup Q| \geq q_w$
        if $req \in$ {TSVal(_, _)} then
            $S \leftarrow \{cv_j \mid p_j \in Q \cup K\}$
        else  %$req \in$ {CV(_, _), preCV(_, _)}
            $S \leftarrow$ read_quorum(CV)
        forall $z = 1..n$ do
            $cv[z] \leftarrow \max\{cv'[z] \mid cv' \in S\}$
        $Q \leftarrow \{p_j \in Q \cup K \mid inc_j \geq cv[j] \vee i = j\}$
    until $|Q| \geq q_w$

function read_quorum($req$)
    $id \leftarrow$ get_unique_id()
    do
        send READ($id, req$) to $\mathcal{R}$
    periodically until received
      {READ_ACK($id, x_j, j$) $\mid p_j \in Q$}
      for some $Q$ such that $|Q| \geq q_r$
    return $\{x_j \mid p_j \in Q\}$

when received READ($id, req$) from $p_j$
    pre: stale = FALSE
    if $req$ = State($inc$) then
        state.cv[$j$] $\leftarrow \max\{$state.cv[$j$]$, inc\}$
    $r \leftarrow$ case $req$ do
            TS: state.ts
            TSVal: (state.ts, state.val)
            preCV($j$): state.pre_cv[$j$]
            CV: state.cv
            State($inc$): state
    send READ_ACK($id, r, i$) to $p_j$

when received WRITE($id$, TSVal($ts, v$), _)
  from $p_j$
    pre: stale = FALSE
    if $ts >$ state.ts then
        (state.ts, state.val) $\leftarrow (ts, v)$
    send WRITE_ACK($id$, state.cv[$i$], state.cv, $i$)
      to $p_j$

when received WRITE($id, req, inc$) from $p_j$
    pre: $req \in$ {CV($j, v$), preCV($j, v$)}
    state.cv[$i$] $\leftarrow \max\{$state.cv[$i$]$, inc\}$
    if $req$ = preCV($j, v$) then
        state.pre_cv[$j$] $\leftarrow$
          $\max\{$state.pre_cv[$j$]$, v\}$
    else
        state.cv[$j$] $\leftarrow \max\{$state.cv[$j$]$, v\}$
    send WRITE_ACK($id$, state.cv[$i$], $\perp, i$) to $p_j$
```

Figure 3 Pseudocode of TEE-Rex at process $p_i \in \mathcal{P}$.

which obtains responses from a read quorum $\{p_1, p_3\}$. Since neither replica is aware of p_4's write, the `read` returns v_0, violating linearizability.

To address the above problem, the set of responses collected by p_4 from a write quorum of replicas must be *crash-consistent* [36]. Let $\rightarrow$ be a partial order on the set of replica events such that $e \rightarrow e'$ if there exists a replica p_i such that e precedes e' at p_i.

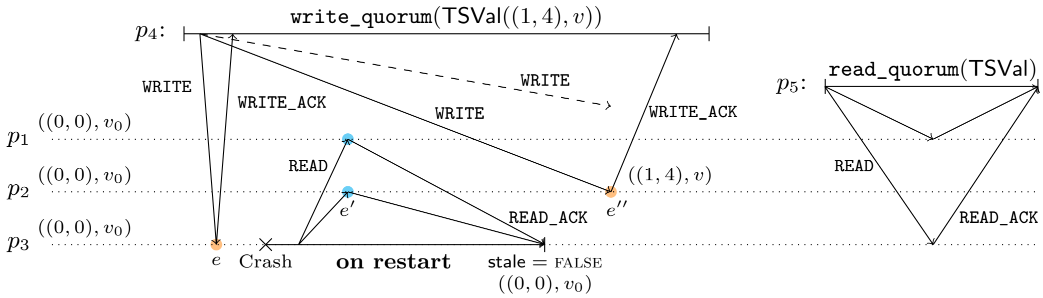

Figure 4 An execution with incorrect recovery.

▶ **Definition 10.** *A set E of* [**send** WRITE_ACK] *events is* ***crash-consistent*** *if for for each event $e \in E$ occurring at replica p_i, if after this event p_i crashes, restarts, and invokes* read_quorum(State)*, then the set E' of* [**send** READ_ACK] *events triggered by the read satisfies $\neg\exists e' \in E'.\ \exists e'' \in E.\ e' \rightarrow e''$.*

For example, consider Figure 4. Let E be the set of [**send** WRITE_ACK] events triggered by the call to $wq =$ write_quorum(TSVal$((1,4),v)$) at p_4 (orange dots), and let E' be the set of [**send** READ_ACK] events triggered by the call to read_quorum(State) during the recovery at p_3 (blue dots). Let $e \in E$ be the event at p_3 sending WRITE_ACK, $e' \in E'$ the event at p_2 sending READ_ACK, and $e'' \in E$ the event at p_2 sending WRITE_ACK. Since $e' \rightarrow e''$, the set E is not crash-consistent.

4.2 Crash-Consistency with Incarnation Numbers and Crash Vectors

To implement crash-consistency, similarly to [25,36], each replica in TEE-Rex maintains two pieces of metadata – an *incarnation number* and a *crash vector*. The former is an integer, initialized to 0, and the latter tracks the highest known incarnation numbers of all other replicas. The crash vector of replica p_i is stored in state.cv, and for convenience, p_i's own incarnation number is stored in state.cv$[i]$. This incarnation number is updated by the recovery procedure at line 9. To explain the basic principles underlying crash-consistency checks in TEE-Rex, we first assume that incarnation numbers written by each replica p_i into state.cv$[i]$ monotonically increase:

▶ **Property 11** (Incarnation Number Monotonicity). *Assume that a replica p_i sets* state.cv$[i]$ *to g at line 9 at time t. Then $g > 0$ and for all times $t' > t$, if p_i sets* state.cv$[i]$ *to g' at line 9 at time t', then $g' > g$.*

This property can be easily ensured assuming that each replica has a built-in monotonic counter that is *never* rolled back, as done in previous work [36]. In §4.3 we show how to ensure this property without such an assumption. The code needed for this is shown in blue and should be ignored when reading this section.

write_quorum(TSVal) implementation. In TEE-Rex, a process receiving a WRITE message piggybacks its incarnation number and a copy of its crash vector on the WRITE_ACK response (line 55). These are used by the implementation of write_quorum(TSVal) to check whether the set of WRITE_ACK responses it receives is crash-consistent.

This check is integrated within the loop executed by write_quorum (line 21). At each iteration of this loop, a WRITE message is (periodically) broadcast to all replicas excluding those in the set Q, which accumulates the replicas whose responses have been already validated as crash-consistent. Once acknowledgments are received from a set K of replicas such that $K \cup Q$ is a write quorum (line 25), the function checks their crash-consistency. The replicas that sent responses that are not crash-consistent are then purged from $K \cup Q$, and the set of remaining replicas is reassigned to Q (line 32).

To identify which replicas p_j in the set $K \cup Q$ sent crash-consistent responses, the crash vectors received in the `WRITE_ACK` messages are combined into a vector cv by taking their entry-wise maximum (line 31). Then the incarnation numbers inc_j in the `WRITE_ACK` messages are compared against $cv[j]$. If $inc_j < cv[j]$, then p_j restarted while our invocation of `write_quorum` was gathering `WRITE_ACK` responses from other replicas, which indicates that p_j's prior response is no longer crash-consistent. As a special case, a replica always treats its own `WRITE_ACK` responses as crash consistent (the $i = j$ disjunct at line 32). If after excluding all replicas whose responses are not crash-consistent, Q contains a write quorum, then `write_quorum` returns (line 33); otherwise, it proceeds to the next iteration of the loop.

Recovery implementation. The recovery procedure executed by replica p_i starts by setting its stale flag to TRUE (line 5), thus preventing p_i from replying to `READ` and `WRITE`(TSVal) messages (lines 41 and 52). The replica then selects a new incarnation number (line 9), which is required to satisfy Property 11, and stores it at a write quorum (line 10; we explain the implementation of `write_quorum`(CV) later). The replica further invokes `read_quorum`(State) to retrieve the states from a read quorum, including both register states and crash vectors (line 11). Finally, the replica reconstructs its state by merging the crash vectors it received and picking the register value with the highest timestamp, and clears the stale flag (lines 12–18).

The `READ`(State) messages sent by the `read_quorum` to retrieve the states (line 11) carry the new incarnation number, and a replica receiving such a message incorporates the new incarnation number into its crash vector (line 43). Thus, the new incarnation number is written to a read quorum of replicas, which intersects any write quorum that may be used by a concurrent invocation of `write_quorum`(TSVal). This ensures that this invocation only accepts a crash-consistent set of responses (Definition 10) and rules out the execution in Example 9, as we now explain.

▶ **Example 12.** Assume that in Figure 4, p_3's incarnation number before the crash is g_1 and its new incarnation number after the recovery is $g_2 > g_1$. Then p_3 will send g_1 in its `WRITE_ACK` response to p_4 (line 55), and g_2 in its `READ`(State(g_2)) request to p_2 (line 11). When p_2 responds to p_3, it will record g_2 in its crash-vector entry for p_3 (line 43). Then p_2 will piggyback this crash vector on its `WRITE_ACK` response to p_4's `WRITE` request (line 55). This will cause p_4 to discard p_3's `WRITE_ACK` response, since it carries a smaller incarnation number (g_1) than p_3's entry in the crash vector received from p_2 (g_2) (line 32).

Note that in the above example, replica p_2 might crash and recover after responding to p_3 but before responding to p_4. When recovering, p_2 will reconstruct its crash vector to a value no lower than what it was before the crash (line 14), thus giving a correct response to p_4. This is ensured by the fact that each recovering replica, such as p_3, writes its incarnation number crash-consistently to a write quorum (line 10), which intersects with a read quorum that a replica such as p_2 uses to reconstruct its state during recovery (line 11).

Thus, a replica p_i writes its incarnation number in the restart handler in two places – first crash-consistently to *some write quorum* (line 10), and then to the *particular read quorum* used to reconstruct its state (line 11). These serve complementary purposes: the former ensures that other replicas can reconstruct the incarnation number of p_i when they restart;

the latter ensures that other replicas can detect when the crash-consistency of their writes can be compromised by a concurrent restart of p_i.

write_quorum(CV) implementation. We now describe the write_quorum implementation for incarnation numbers (line 10), which happens to be subtle. We could implement write_quorum(CV) in the exact same way as write_quorum(TSVal), by using crash vectors piggybacked on WRITE_ACK messages to check for crash-consistency of a write; this was the approach taken in [36]. Unfortunately, the resulting algorithm would not be live cases where liveness must be ensured to match the lower bound of Theorem 7.

▶ **Example 13.** Let $n = 5$, $d = 1$ and $c = 3$, so that $n \geq d + c + 1$. Consider an execution of the above version of the algorithm where p_1 crashes permanently, p_2 and p_3 crash and restart once at the beginning of the execution, and the remaining replicas never crash. This execution is valid under CRR-D$(1, 3, _)$ (case 2 in Definition 4). But p_2 and p_3 would not be able to complete the recovery in it, because their calls to write_quorum(CV) would wait for $q_w = n - d = 4$ responses (line 25). However, only active replicas would respond to WRITE messages (line 52), and there are only 2 such replicas.

To rule out such executions, in our algorithm a replica accepts a WRITE message for a write to state.cv even if its stale flag is set (line 56). In this case the replica cannot piggyback its crash vector on the WRITE_ACK response (as in line 55): the replica may have lost the crash vector upon a restart and has not yet reconstructed it (line 14). Hence, the replica puts a dummy value $\perp$ in place of a crash vector (line 63). The write_quorum function then checks crash-consistency for writes of incarnation numbers differently from register writes: after it receives enough WRITE_ACK responses, it calls read_quorum(CV) to read the crash vectors explicitly (line 29). Replicas only reply to READ requests sent by this function when their stale flag is cleared (line 41), and thus they have valid crash vectors in their state.

In the absence of further crashes in Example 13, p_2 and p_3 can complete the recovery as follows: they first collect WRITE_ACKs from $q_w = n - d = 4$ replicas that are up (p_2, p_3, p_4, p_5) and then read crash vectors from $q_r = d + 1 = 2$ replicas that are active (p_4, p_5). We can show that TEE-Rex is correct assuming Incarnation Number Monotonicity.

▶ **Theorem 14.** *If incarnation numbers assigned at line 9 satisfy Property 11, then the algorithm in Figure 3 excluding the highlighted lines satisfies the conditions of Theorem 6.*

We defer the proof of the theorem to [26, §B]. As in §3.1, we show the safety of the algorithm by first establishing the Real-Time Property (Property 3). We next explain how to ensure Incarnation Number Monotonicity.

4.3 Implementing Incarnation Numbers without Non-Volatile Storage

Since in the presence of rollbacks the replicas cannot rely on non-volatile storage to implement monotonically growing incarnation numbers, TEE-Rex relies on a distributed mechanism for this purpose, shown in blue in Figure 3. This mechanism extends the one used to store incarnation numbers in state.cv (line 10) described in the previous section. Note that a replica cannot restore its previous incarnation number by just reading it using read_quorum(CV): if the replica crashed before completing the write_quorum(CV) at line 10, the read_quorum is not guaranteed to restore its latest pre-crash incarnation number, thus violating Property 11.

To address this problem, we adopt a two-phase approach. In addition to cv, the state of each replica includes a vector pre_cv. Before assigning its new incarnation number to state.cv$[i]$ (line 9), a replica p_i first writes it to state.pre_cv$[i]$ at a crash-consistent write quorum of replicas (line 8). Upon a restart, a replica p_i restores its previous incarnation number as a maximum of responses returned by `read_quorum`(preCV(i)) and computes the new incarnation number by incrementing the result (lines 6–7). Then the intersection between the quorums used by `write_quorum`(preCV) (line 8) and `read_quorum`(preCV) (line 6) helps ensure Property 11.

There is, however, a subtlety. Recall that in our algorithm a replica has to reply to a `WRITE` message for a write to state.cv even while it restarting (line 56), and the same should hold for writes to state.pre_cv: this is necessary to avoid deadlocks like the one in Example 13. When incarnation numbers are computed as described above, a restarting replica thus needs to reply to a `WRITE` even before it computed its incarnation number at line 9. This poses a dilemma: which incarnation number should the replica include into its `WRITE_ACK` message (line 55) to allow other replicas to validate the crash-consistency of their writes to state.cv and state.pre_cv? To address this challenge, when a replica p_j executing `write_quorum`(CV) or `write_quorum`(preCV) sends a `WRITE` message to another replica p_i, it piggybacks the maximum incarnation number of p_i known to it on this message (line 24). A replica p_i receiving a `WRITE` request for state.pre_cv[i] or state.cv[i] then adopts this incarnation number (line 58) and uses it in its `WRITE_ACK` reply.

In [26, §B] we prove that, even though the above mechanism may require processes to *temporarily* adopt stale incarnation numbers, it does ensure Incarnation Number Monotonicity (Property 11). Thus, given Theorem 14, the overall TEE-Rex algorithm is correct. Here we illustrate the operation of the incarnation number implementation on an example.

▶ **Example 15.** Let $n = 5$, $d = 1$ and $c = 3$, so that $n \geq d + c + 1$. Consider an execution where p_1 crashes permanently, p_2 and p_3 are eventually up, and the remaining replicas never crash. This execution is valid under CRR-D$(1, 3, _)$ (case 2 in Definition 4). At the beginning of the execution, p_2 and p_3 crash, restart, and become active again, so that p_4 and p_5 have state.cv[2] = state.cv[3] = 1. Suppose now that p_2 and p_3 crash and restart again.

To recover, each of p_2 and p_3 determines its previous incarnation number 1 (line 6) and computes a new incarnation number 2, which it tries to store crash-consistently in state.pre_cv at a write quorum (line 8). Since p_2 does not have any information about the incarnation number of p_3, its first `WRITE` message to p_3 carries 0 as the incarnation number (line 24), which p_3 includes into its `WRITE_ACK` reply. The `read_quorum`(CV) that p_2 invokes after this (line 29) fetches 1 for state.cv[3] from a read quorum $\{p_4, p_5\}$, so the `WRITE_ACK` from p_3 is discarded (line 32). The same happens at p_2 with the `WRITE_ACK` it receives from p_3.

Each of p_2 and p_3 then includes 1 into the next `WRITE` message it sends to the other replica (line 24), which the latter adopts as its incarnation number and includes into its `WRITE_ACK` reply (line 58). Once p_2 receives the `WRITE_ACK` from p_3, it calls `read_quorum`(CV) once more (line 29) and validates the `WRITE_ACK` as crash-consistent, finishing the execution of `write_quorum`(preCV(2, 2)). Replica p_2 then sets state.cv[2] to its new incarnation number 2 (line 9) and completes the rest of the recovery, including `write_quorum`(CV(2, 2)).

Assume that now p_3 receives the `WRITE_ACK` from p_2 and calls `read_quorum`(CV) to validate it (line 29). This read yields 2 for state.cv[2], so p_3 discards the `WRITE_ACK` and sends another `WRITE` to p_2 with incarnation number 2 (line 24). Now p_2 responds with a `WRITE_ACK` carrying its incarnation number 2, which it also stores locally. When p_3 receives this `WRITE_ACK`, it performs another `read_quorum`(CV), validates the `WRITE_ACK` as crash-consistent and completes `write_quorum`(preCV(3, 2)). It then sets state.cv[3] to its new incarnation number 2 (line 9) and completes the rest of the recovery. In the end, both p_2 and p_3 successfully recover with a higher incarnation number, satisfying Property 11.

5 Related Work

The classical crash-recovery failure model assumed that every process is equipped with durable storage that cannot be rolled back [22]. This assumption was lifted by Aguilera et al. [4], who analyzed consensus solvability as a function of the number of processes with and without durable storage. The same model was also used by Guerraoui et al. [17] to derive tight bounds for a reliable register construction. In these models the processes are assumed to know whether they are equipped with durable storage, which they can trust to be incorruptible. In contrast, in our CRR model a recovering process cannot trust its non-volatile storage to be up-to-date, leading to different resilience bounds. Furthermore, in the register implementation of Guerraoui et al. a restarted replica is considered up-to-date once it accepts a single `WRITE` request. As we show in [26, §C], this leads to a safety violation.

Dinis et al. [13] proposed a rollback-recovery (RR) model to capture rollback attacks in TEE-based systems. This model is weaker than CRR as it disallows permanent process crashes. Furthermore, the register implementation of Dinis et al. uses ordinary quorums for writes, without crash-consistency checks. As we show in [26, §D], this leads to a safety violation similar to the one in Example 9. The implementation also does not guarantee wait-freedom for reads. In contrast, our CRR framework captures the full spectrum of failures in TEE-based systems and enables developing correct solutions for these environments.

The crash-recovery failure model where the processes do not have access to incorruptible durable storage was assumed by Chandra et al. [10], Kończak et al. [27], and Liskov et al. [30] in the context of their efforts to develop practical variants of Paxos [29] and viewstamped replication [38]. The proposed solutions, however, did not use crash-consistent quorums for storing their state, and as a result, were shown in [36] to violate safety.

Jehl et al. [25] proposed a versioning scheme similar to crash vectors that allows quickly replacing a failed replica in Paxos. This technique was subsequently generalized to the notion of crash-consistent quorums by Michael et al. [36], who also demonstrated how it can be used to implement a recoverable atomic register. However, the safety of these versioning schemes critically depends on the replicas' ability to track their incarnations across restarts. In turn, supporting this capability in TEE-based systems requires hardware-based persistent counters. These counters are implemented using flash memory, resulting in poor write performance and quick wear-out [35]. Furthermore, they are not universally supported by TEE manufacturers, and have been recently deprecated by Intel SGX [23].

Although the technical report version [37] of the work by Michael et al. suggests that such incarnation numbers can be supported by means of a distributed mechanism, it does not provide a full implementation. In addition, while the recoverable register implementation of [36,37] is always safe, its liveness under the static failure models requires $n = 2c+1$, which is strictly worse than our bound of $n \geq d+c+1$.

Persistent monotonic hardware counters have been demonstrated in [40] to be a powerful mechanism to guard against rollback attacks. In TEE-based systems they can be used together with sealing (§1) to validate state freshness upon restart [12]. Unfortunately, these solutions inherit the drawbacks of persistent counters we explained above.

ROTE [35] guards against rollbacks using a rollback-resistant distributed counter. Its implementation relies on a two-round protocol where a new value of the counter is first written to a write quorum, and then read back from the *same* write quorum to validate that it was stored reliably. Although this solution is safe, it does not ensure liveness, as the same write quorum cannot be guaranteed to be available two times in a row under failures and asynchrony. Engraft [42] uses a similar protocol, so is subject to the same liveness issue.

References

1 Ittai Abraham, Gregory V. Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk Paxos: optimal resilience with Byzantine shared memory. *Distributed Comput.*, 18(5), 2006.

2 Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. Good-case latency of Byzantine broadcast: A complete categorization. In *Symposium on Principles of Distributed Computing (PODC)*, 2021.

3 Yehuda Afek, David S. Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared objects. *J. ACM*, 42(6), 1995.

4 Marcos K Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Comput.*, 13(2), 2000.

5 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. In *Symposium Principles of Distributed Computing (PODC)*, 1990.

6 Hagit Attiya, Hyun Chul Chung, Faith Ellen, Saptaparni Kumar, and Jennifer L. Welch. Simulating a shared register in an asynchronous system that never stops changing. In *Symposium on Distributed Computing (DISC)*, 2015.

7 Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on steroids: SGX-based high performance BFT. In *European Conference on Computer Systems (EuroSys)*, 2017.

8 Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. In *Conference on Dependable Systems and Networks (DSN)*, 2017.

9 Dhruva R. Chakrabarti, Hans-Juergen Boehm, and Kumud Bhandari. Atlas: leveraging locks for non-volatile memory consistency. In *Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2014.

10 Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Symposium on Principles of Distributed Computing (PODC)*, 2007.

11 Gregory Chockler, Nancy Lynch, Sayan Mitra, and Joshua Tauber. Proving atomicity: An assertional approach. In *Symposium on Distributed Computing (DISC)*, 2005.

12 Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

13 Baltasar Dinis, Peter Druschel, and Rodrigo Rodrigues. RR: A fault model for efficient TEE replication. In *Network and Distributed System Security Symposium (NDSS)*, 2023.

14 Dan Dobre, Rachid Guerraoui, Matthias Majuntke, Neeraj Suri, and Marko Vukolic. The complexity of robust atomic storage. In *Symposium on Principles of Distributed Computing (PODC)*, 2011.

15 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for Byzantine agreement. *J. ACM*, 32(1), 1985.

16 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2), 1988.

17 Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Jim Pugh. The collective memory of amnesic processes. *ACM Trans. Algorithms*, 4(1), 2008.

18 Rachid Guerraoui and Marko Vukolic. How fast can a very robust read be? In *Symposium on Principles of Distributed Computing (PODC)*, 2006.

19 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1), 1991.

20 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *International Conference on Distributed Computing Systems (ICDCS)*, 2003.

21 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.

22 Michel Hurfin, Achour Mostéfaoui, and Michel Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Symposium on Reliable Distributed Systems (SRDS)*, 1998.

23 Intel Corporation. End of life for Intel SGX monotonic counter service. https://www.intel.com/content/www/us/en/support/articles/000057968/software/intel-security-products.html, 2023. Accessed: 2025-02-09.

24 Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, 45(3), 1998.

25 Leander Jehl, Tormod Erevik Lea, and Hein Meling. Replacement: Decentralized failure handling for replicated state machines. In *Symposium on Reliable Distributed Systems (SRDS)*, 2015.

26 Sadegh Keshavarzi, Gregory Chockler, and Alexey Gotsman. TEE is not a healer: Rollback-resistant reliable storage (extended version). *arXiv*, 2505.18648, 2025.

27 Jan Kończak, Nuno Santos, Tomasz Żurkowski, Paweł T. Wojciechowski, and André Schiper. JPaxos: State machine replication based on the Paxos protocol. Technical Report EPFL-REPORT-167765, EPFL, 2011.

28 Leslie Lamport. On interprocess communication. *Distributed Comput.*, 1(2), 1986.

29 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.

30 Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT CSAIL, 2012.

31 Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. Scalable Byzantine consensus via hardware-assisted secret sharing. *IEEE Trans. Computers*, 68(1), 2019.

32 Nancy A Lynch and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Comput.*, 1(1), 1986.

33 Nancy A. Lynch and Alexander A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Symposium on Distributed Computing (DISC)*, 2002.

34 Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal Byzantine storage. In *Symposium on Distributed Computing (DISC)*, 2002.

35 Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback protection for trusted execution. In *USENIX Security Symposium (USENIX Security)*, 2017.

36 Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. Recovering Shared Objects Without Stable Storage. In *Symposium on Distributed Computing (DISC)*, 2017.

37 Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. Recovering shared objects without stable storage (extended version). Technical Report UW-CSE-TR-17-10-01, University of Washington, 2017.

38 Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. *Symposium on Principles of Distributed Computing (PODC)*, 1988.

39 Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2), 1980.

40 Raoul Strackx and Frank Piessens. Ariadne: A minimal approach to state continuity. In *USENIX Security Symposium (USENIX Security)*, 2016.

41 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. Efficient Byzantine fault-tolerance. *IEEE Trans. Computers*, 62(1), 2013.

42 Weili Wang, Sen Deng, Jianyu Niu, Michael K. Reiter, and Yinqian Zhang. ENGRAFT: Enclave-guarded Raft on Byzantine faulty nodes. In *Conference on Computer and Communications Security (CCS)*, 2022.

43 Qianyu Yu, Giuliano Losa, and Xuechao Wang. TetraBFT: Reducing latency of unauthenticated, responsive BFT consensus. In *Symposium on Principles of Distributed Computing (PODC)*, 2024.