# Providing Stable Storage for the Diskless Crash-Recovery Failure Model

Ellis Michael Dan R. K. Ports Naveen Kr. Sharma Adriana Szekeres
{emichael, drkp, naveenks, aaasz}@cs.washington.edu
*University of Washington*

Technical Report UW-CSE-16-08-02

August 25, 2016

## Abstract

Many classic protocols in the fault tolerant distributed computing literature assume a Crash-Fail model in which processes either are up, or have crashed and are permanently down. While this model is useful, it does not fully capture the difficulties many real systems must contend with. In particular, real-world systems are long-lived and must have a recovery mechanism so that crashed processes can rejoin the system and restore its fault-tolerance. When processes are assumed to have access to stable storage that is persistent across failures, the Crash-Recovery model is trivial. However, because disk failures are common and because having a disk on a protocol's critical path is often performance concern, diskless recovery protocols are needed. While such protocols do exist in the state machine replication literature, several well-known protocols have flawed recovery mechanisms. We examine these errors to elucidate the problem of diskless recovery and present our own protocol for providing *virtual stable storage*, transforming any protocol in the Crash-Recovery with stable storage model into a protocol in the Diskless Crash-Recovery model.

## 1 Introduction

Distributed algorithms are needed to build reliable services out of unreliable processes. To effectively support long-lived systems, these algorithms must be both robust to process and communication failures and able to help processes recover from crashes. That is, algorithms must provide provable guarantees in both the Crash-Stop and Crash-Recovery models.

Prior algorithms generally assume a Crash-Recovery model with *stable storage*, e.g., a disk attached to each process whose contents are never lost. In practice, this assumption does not always hold. Because writing data synchronously to disk incurs a high performance cost, many systems now eschew persistent disk writes in favor of replicated in-memory storage. Furthermore, disks can become corrupted or totally fail, and real-world systems must cope with these challenges.

To address this need, we introduce a *Diskless Crash-Recovery* model: processes lose their state on failures but can use a recovery protocol to regenerate their state upon recovery. We formally define this model and compare it to traditional Crash-Stop and Crash-Recovery models.

The central question this paper answers is: *How and when should an algorithm for Crash-Stop or Crash-Recovery models be transformed into one for Diskless Crash-Recovery?* It is well known that a correct algorithm for the asynchronous Crash-Stop model can be converted to an algorithm for the Crash-Recovery model simply by recording each state transition to stable storage. Is a similar transformation possible to a Diskless Crash-Recovery model? Intuitively, we should be able to construct a diskless algorithm by replacing writes to stable storage with writes to a quorum of processes so that data remains available upon recovery.

Several algorithms have attempted to provide ad-hoc solutions to this problem in the context of specific protocols; none has been general purpose. In particular, diskless recovery is of great interest for state machine replication, which demands a reliable system that must be both long-lived and tolerant of node and disk failures. To our knowledge, no current algorithm correctly

handles recovery in the diskless model. We examine three prior algorithms and demonstrate how they violate the safety conditions of state machine replication under certain failure conditions.

This paper introduces the first general purpose algorithm for transforming Crash-Stop or Crash-Recovery algorithms to the Diskless Crash-Recovery model. It provides a *virtual stable storage* abstraction with the same interface as a disk but constructed using distributed volatile storage. This algorithm provides *durability* (i.e., any data written to the virtual stable storage is readable later) and *liveness* (i.e., reads and writes eventually complete). We prove that our algorithm guarantees safety in the Diskless Crash-Recovery model in all cases and guarantees termination given reasonable assumptions about failure patterns.

The most closely related work to ours is Aguilera et al.'s study of fault tolerance in an earlier Crash-Recovery model without stable storage [1]. That work concludes that consensus is solvable in their model only when at least one process never crashes – an unrealistic assumption for long-lived systems, as the authors themselves admit. Our work provides a solution even when no process remains always up. The key differentiator is that our work permits nodes to run a *recovery protocol* after crashes rather than being forced to immediately resume the normal-case protocol.

# 2 Models and Definitions

This section formally defines three models: the classic Crash-Stop model, a Crash-Recovery model that uses stable storage, and the Diskless Crash-Recovery model without stable storage.

In each case, we consider a distributed system of a fixed set of $n$ processes, with IDs $1, \ldots,$ n. Each process is modeled as an I/O automaton [11] that takes input or an internal action, produces output, and transitions between states. A distributed execution happens in discrete timesteps, during each of which, one (or more) process(es) takes a step. Processes communicate with each other by sending messages through a complete, asynchronous network. This means that messages can be lost, duplicated, or reordered arbitrarily – but not modified – by the network. However, we assume the network cannot duplicate or drop messages an infinite number of times (i.e., the network is fair-lossy).

## 2.1 Crash-Stop Model

The *Crash-Stop* (CS) model assumes that processes can fail by crashing and that crashes are permanent; a failed node stops executing the algorithm forever and stops communicating with other nodes. We say a process is UP if it is executing protocol steps and performing actions. If it crashes, it transitions to the DOWN state. Once DOWN, a process no longer accepts messages that are sent to it and forever remains in the DOWN state.

## 2.2 Crash-Recovery Model with Local Stable Storage

In the *Crash-Recovery* model (CR), a process can *recover* after crashing and resume executing the algorithm. As in the CS model, a process in this model is assumed to be either UP or DOWN, but it can also transition between the two states an infinite number of times. A process that is DOWN can *recover* and transition back to the UP state.

We want this model to capture the distinction between the volatile state of the system and stable storage. As a result, the automaton does not recover in the same state it was in before the crash. Rather, it can access stable storage that it must manage explicitly. We abstract access to stable storage using an additional internal automaton action, $\mathsf{WRITE}(X)$, that persists $X$ on stable storage. If the action completes successfully, $X$ will always be seen by the automaton irrespective of the number of crashes. If the process crashes while $\mathsf{WRITE}(X)$ is pending, $X$ may or may not be persisted.

When a process restarts in this model, it can recover state by using another internal action, READ, that returns the *set* of all values $X$ that were persisted using the $\mathsf{WRITE}(X)$ action in the past. Our choice of a set of values as the interface to READ is somewhat arbitrary. We could instead provide a log interface or a single atomically-updated value; these options are equivalent to one another, in that each can trivially be constructed from the others.

## 2.3 Diskless Crash-Recovery: Crash-Recovery without Local Stable Storage

We focus on the *Diskless Crash-Recovery* (DCR) model, where nodes have no access to stable storage. As in the CR model, a process can recover after crashing and resume execution. However, unlike CR, DCR has no READ or WRITE actions.

Upon recovery, a process loses all state except its unique process ID in $\{1, 2, ..., n\}$. However, processes can read from a local clock to get a number guaranteed to be larger than any previous number read by any previous incarnation of that process. Importantly, this makes

it possible to distinguish between different incarnations of the same processes.[1]

A recovering process must bring itself to a state that lets it resume its computation without violating any guarantee of the algorithm. To do so, a process that recovers runs a distinct *recovery protocol*. This protocol can communicate with other processes to recover previous state. Once the recovery protocol terminates, the automaton changes its internal status to operational and resumes execution of its normal protocol.

We describe a process that is UP as RECOVERING if it is running its recovery protocol and OPERATIONAL otherwise. This distinction makes it possible to state failure bounds in terms of the number of operational processes, e.g., that no more than half of the processes can be either down or recovering at any moment. Our definition matches the design of existing protocols (e.g., Viewstamped Replication [10]).

One important property that protocols in the DCR model should satisfy is *recovery termination*, which requires a recovering process to eventually complete recovery and become OPERATIONAL, as long as it does not crash again. We are thus not interested in trivial solutions, i.e., those where recovering processes declare themselves to be permanently DOWN and never again participate in the normal protocol.

# 3 Model Transformations

Many algorithms solve problems in the CS failure model. When can these algorithms be applied to solving the same problems in the CR or DCR models?

**Crash-Stop to Crash-Recovery with Stable Storage.** Transforming CS protocols to the CR model is trivial when local stable storage is available. This can be done by recording every state transition to disk before performing output actions. Equivalently, the automaton can log every message and input action received to stable storage and replay these in order on recovery.

Any asynchronous CS algorithm transformed in this way provides the same guarantees in the CR model. Because the automaton recovers in the same state it was in before crashing, the only difference is that it does not process any messages that it received while it was down. Asynchronous algorithms are inherently robust to these *omission failures* [3] because they can handle messages that the network drops or delays.

**Towards Diskless Crash Recovery.** This paper poses the question, *How can a protocol from the Crash-Stop or Crash-Recovery model be correctly transformed to one that works in the Diskless Crash-Recovery model?* Specifically, can we implement a *virtual stable storage* abstraction in the DCR model that provides the same READ/WRITE() interface defined in CR? In the DCR model, a process can persist state only by replicating on other processes in the system. Hence, any READ/WRITE(X) action must communicate with other processes in the system to provide the same guarantees even in the presence of concurrent failures. We begin by defining the notion of virtual stable storage:

**Virtual Stable Storage.** We say that an algorithm correctly implements *virtual stable storage* in the DCR model if it provides READ and WRITE(X) primitives with the following properties:

- **Persistence**: A successful READ outputs the set of *all* objects the process successfully persisted through a prior invocation of WRITE(X).
- **Termination**: Every READ and WRITE(X) at a process eventually succeeds unless the process crashes.

# 4 State Machine Replication in Diskless Crash-Recovery

Providing stable storage in the DCR model seems superficially straightforward – simply replace each write to disk with a write to a majority of replicas. However, there are surprisingly subtle challenges; if handled naïvely, delayed messages from prior incarnations of a process can lead to the illusion that a write has been persisted when, in fact, it could still be lost.

As evidence of its subtlety, several protocols have attempted, and failed, to provide diskless crash recovery in the context of a specific problem, state machine replication. State machine replication (SMR) – a classic problem that lies at the core of many critical deployed distributed systems [2, 7, 9, 10, 13, 15] – highlights the relevance of diskless recovery: SMR deployments are long-lived and must be able to handle node crashes and recoveries, including ones where stable storage is lost.

We analyze 3 state machine replication protocols for the DCR model: Viewstamped Replication [10], Paxos Made Live [2], and JPaxos [6]. In each case, we show that these protocols do not completely recover the state

[1] The algorithm we present in Section 5 could be easily adapted to work when processes can only generate a unique number instead of a monotonically increasing one. We assume a monotonic clock only for simplicity of exposition.

of a failed node under certain scenarios, a situation that can lead to user-visible violations of protocol correctness. To our knowledge, no prior work has provided a correct recovery protocol for this problem.

This section provides a detailed case study of correctness problems in Viewstamped Replication and an overview of the equivalent problem in Paxos Made Live and JPaxos. A more complete description, including detailed traces of the failure in all three protocols, is available in Appendix A.

### 4.1 Definitions

The SMR approach models a replicated service as a state machine. Clients submit requests to the service, which runs a consensus protocol (e.g., Multi-Paxos or Viewstamped Replication) to establish a global order of requests; the replicas then execute that request and respond to the client. A correct SMR protocol provides *linearizability* [4]: clients receive responses as though their requests had been executed by a single system in a serial order; if operation A receives a response before B is invoked, A must precede B in that serial order.

SMR is well known to be equivalent to repeated instances of consensus, or atomic broadcast. We choose to frame the problem as SMR rather than consensus because the former implies a long-lived system consistent with our DCR model. In particular, single-instance consensus admits some solutions (such as requiring some fraction of nodes to be up for the duration of the protocol [1]) that are not reasonable in the long-lived SMR model.

### 4.2 Viewstamped Replication

Viewstamped Replication (VR) is a classic SMR algorithm. The original presentation by Oki and Liskov in 1988 [13] used stable storage for certain parts of the replica state. VR always guarantees sequence agreement and integrity, and it guarantees liveness as long as there is sufficient network synchrony and no more than $f$ out of $2f + 1$ replicas are failed. A subsequent version [10] revised the protocol to eliminate the need for stable storage; we demonstrate that this version does not correctly handle all loss-of-state failures.

VR is a leader-based algorithm: the system moves through a series of numbered views, in which one node is designated as the leader. VR uses 2 protocols. During normal case execution, the leader assigns sequence numbers to incoming client requests, sends PREPARE messages to replicas, and executes the operation once it has received replies from a majority. When the leader is suspected to have failed, a *view change* protocol replaces the leader with a new one. Replicas increment their view numbers, stop processing requests in the old view, then send the new leader a VIEW-CHANGE message with their log of operations. The new leader begins processing only when it receives VIEW-CHANGE messages from a majority of replicas, ensuring that it knows about all operations successfully completed in prior views. These two protocols are equivalent to the two phases of Paxos.

**View Change Invariant.** In VR, an important invariant is that each replica's view number increases monotonically: *once a replica sends a* VIEW-CHANGE *message for view v, it never returns to a lower view*. This is important for correctness because it implies that replicas cannot commit to new operations in prior views once they have sent a VIEW-CHANGE message to the new leader. Without this invariant, a new leader cannot be guaranteed to know about all operations completed by previous leaders and would thus violate linearizability.

**Recovery in VR.** Ensuring that nodes in VR can recover from failures requires providing a recovery procedure. This procedure must ensure that the view change invariant continues to hold, i.e., that *each replica recovers in a view number at least as high as the view number in any* VIEW-CHANGE *message it has ever sent.*

The original version of VR [13] achieved this invariant through stable storage. This protocol logged view numbers to stable storage during view changes, but it eschewed the use of stable storage in normal operations because writing every operation to disk is slow. It used a recovery protocol to recover all other state, including the set of committed operations.

A later version, "VR Revisited" [10], claimed to provide a completely diskless mode of operation. It used a recovery protocol and an extension to the view change protocol to, in essence, replace each write to disk with communication with a quorum of nodes. We show below that this protocol is insufficient to ensure continued correctness of the system.

VR Revisited's recovery protocol is straightforward: the recovering replica sends a RECOVERY message to all other replicas.[2] If not recovering or in the middle of a view change, every other node replies with a RECOVERY-RESPONSE containing its view number;

[2] This message contains a unique nonce to distinguish responses from different recoveries if a node recovers more than once.

the leader also includes its log of operations. Once the recovering replica has received a quorum of responses with the same view, including one from that view's leader, it updates its state with the information in the log.

VR Revisited adds another phase to the view change protocol. When nodes determine a view change is necessary, they increment their view number, stop processing requests in the old view, and send a START-VIEW-CHANGE message to all other replicas. Only when replicas receive START-VIEW-CHANGE messages from a quorum of replicas do they send their VIEW-CHANGE message to the new leader and proceed as in the original protocol. The additional phase is intended to serve as the equivalent of a disk write, ensuring that replicas do not commit to a new view until a majority of them becomes aware of the intended view change. Together with the recovery protocol, the added phase aims to prevent violation of the View Change Invariant by ensuring that a crashed replica recovers in a view at least as high as any VIEW-CHANGE message it has sent.

By itself, however, this approach is not sufficient. The problem of persistence has only shifted a layer: rather than a node "forgetting" that it sent a VIEW-CHANGE message on a crash, it can forget that it sent a START-VIEW-CHANGE. That is, consider the following case with three nodes:

- Node $A$ initiates a view change by sending START-VIEW-CHANGE $v+1$, but this message is delayed.
- $A$ crashes and recovers; it receives RECOVERY-RESPONSEs from $B$ and $C$. Because they have not seen the START-VIEW-CHANGE message, $A$ recovers to view $v$.
- $B$ receives $A$'s START-VIEW-CHANGE and sends START-VIEW-CHANGE $v+1$. Because it has a quorum of these messages (from $A$ and itself), it sends VIEW-CHANGE $v+1$, but this message is delayed.
- $B$ crashes, recovers, and receives RECOVERY-RESPONSEs from nodes $A$ and $C$ during recovery. These nodes are in view $v$ and have no knowledge of the view change, so $B$ recovers to view $v$.

$B$ has now sent a VIEW-CHANGE message but has lost all knowledge of that fact, violating the View Change Invariant.

This leads to a violation of linearizability. $B$ can now accept a new operation $o$ in view $v$ by sending PREPARE-OK messages to the leader (say it is $A$). This operation can succeed without $C$ learning of the operations, because $A$ and $B$ together form a quorum. If $C$ suspects that $A$ has failed and initiates a view change, it can proceed using only its own log and the delayed VIEW-CHANGE message from $B$. Neither log contains $o$, so this operation will not be visible to future clients.

Appendix A provides a complete trace of a linearizability violation in the VR Revisited protocol.

### 4.3 Other Protocols: Paxos Made Live and JPaxos

Viewstamped Replication is not the only protocol that attempts to provide diskless recovery for SMR. The same type of correctness problem exists in two other protocols, Paxos Made Live [2] and JPaxos [6], and we briefly review their recovery approaches here; a full description is in Appendix A.

**Paxos Made Live** [2] is Google's implementation of the Multi-Paxos protocol. To handle corrupted disks, it lets a replica rejoin the system without its previous state and runs an (unspecified) recovery protocol to restore the application state. The recovering replica must then wait to observe a full instance of successful consensus before participating. This step successfully prevents the replica from accepting multiple values for the same instance (e.g., one before and one after the crash). However, it does not prevent the replica from sending different *promises* (i.e., view change commitments) to potential new leaders, which can lead to a new leader deciding a new value for a prior successful instance of consensus.

**JPaxos** [6], a hybrid of Multi-Paxos and VR, provides a variety of deployment options, including a diskless one. Nodes in JPaxos maintain an *epoch vector* that tracks which nodes have crashed and recovered to discard lost promises made by prior incarnations of recovered nodes. However, like VR and PML, this approach encounters the same problem at a different level: certain failures during node recovery can cause the system to lose state and violate safety properties.

## 5 Providing Stable Storage in Diskless Crash-Recovery

In this section we present a protocol that provides the *virtual stable storage* abstraction in the Diskless Crash-Recovery model along with its correctness proof. The protocol implementation is shown in Algorithm 1.

**Algorithm 1** Single reader, single writer non-atomic set in Diskless Crash-Recovery

**Permanent Local State:**
- $n \in \mathbb{N}^+$ ▷ Number of processes
- $i \in [1,\dots,n]$ ▷ Process number

**Volatile Local State:**
- $v \leftarrow [\bot$ **for** $i \in [1,\dots,n]]$ ▷ Crash vector
- $op \leftarrow false$ ▷ Operational flag
- $R \leftarrow \{\}$ ▷ Acquire reply set
- $w \leftarrow \bot$ ▷ Value being written
- $S \leftarrow \{\bot\}$ ▷ Local set

1: **upon** SYSTEM-INITIALIZE
2: $op \leftarrow true$
3: **end upon**

4: **upon** RECOVER
5: $v[i] \leftarrow$ READ-CLOCK
6: ACQUIRE-QUORUM($\bot$)
7: $op \leftarrow true$
8: **end upon**

9: **procedure** WRITE($val$)
10: **guard:** $op$
11: ACQUIRE-QUORUM($val$)
12: **end procedure**

13: **function** ACQUIRE-QUORUM($val$)
14: $w \leftarrow val$
15: $R \leftarrow \{\}$
16: Add $val$ to $S$
17: $m \leftarrow \langle$ACQUIRE, $S\rangle$
18: **for all** $j \in [1,\dots,n]$ **do**
19: SEND-MESSAGE($m$, $j$)
20: **end for**
21: Wait until $|R| > n/2$
22: **end function**

23: **procedure** READ
24: **guard:** $op$
25: **return** $S - \{\bot\}$
26: **end procedure**

27: **function** SEND-MESSAGE($m$, $j$)
28: $m.f \leftarrow i$ ▷ Sender
29: $m.v \leftarrow v$
30: Send $m$ to process $j$
31: **end function**

32: **function** DISCARD-OLD-REPLIES
33: **while** $\exists m \in R$ **where**
34: $m.v[m.f] < v[m.f]$ **do**
35: Remove $m$ from $R$
36: SEND-MESSAGE($\langle$ACQUIRE, $S\rangle$, $m.f$)
37: **end while**
38: **while** $\exists m, m' \in R$ **where**
39: $m.f = m'.f \wedge m \neq m'$ **do**
40: Remove $m$ from $R$
41: **end while**
42: **end function**

43: **upon** receiving $\langle$ACQUIRE$\rangle$, $m$
44: **guard:** $op$
45: $v \leftarrow v \sqcup m.v$
46: $S \leftarrow S \cup m.S$
47: $m' \leftarrow \langle$ACQUIRE-REP, $S\rangle$
48: SEND-MESSAGE($m'$, $m.f$)
49: **end upon**

50: **upon** receiving $\langle$ACQUIRE-REP$\rangle$, $m$
51: **guard:** $m.v[i] \geqslant v[i] \wedge w \in m.S$
52: $v \leftarrow v \sqcup m.v$
53: $S \leftarrow S \cup m.S$
54: Add $m$ to $R$
55: DISCARD-OLD-REPLIES
56: **end upon**

## 5.1 Overview

The protocol we propose implements a single reader, single writer, fault-tolerant *non-atomic set* that provides the READ/WRITE interface and virtual stable storage properties presented in Section 3. Adapted from the definition of regular registers [8], a non-atomic set offers weaker guarantees than a linearizable one. It guarantees that a READ returns a set containing all previously successfully written values, but it makes no other guarantees. Importantly, the process is allowed only to write values to the set, never to remove them. We choose these weaker guarantees for simplicity, so we can focus on the main problem – how to correctly obtain diskless persistence in the presence of crashes and recoveries.

Stronger properties can be implemented on top of this basic interface. For instance, this protocol could easily be adapted to provide a multi-reader, multi-writer non-atomic set. However, to guarantee that a READ returns *all* previously written values, a process would have to read from a (simple, not necessarily consistent) quorum instead of its local set. A multi-reader, multi-writer set would be a powerful abstraction upon which to readily build other data structures (e.g., the shared log of a replicated state machine). Also, it is worth noting that concurrent instances of this single reader, single writer set can be run to give *all* processes access to virtual stable storage.

While the virtual stable storage protocol is always safe, the *termination* of WRITE and recovery are guaranteed only under certain assumptions. Section 5.3 specifies these assumptions and proves protocol correctness.

## 5.2 Protocol Description

We now describe the protocol, which is presented as pseudo-code in Algorithm 1. We present the algorithm using a modified I/O automaton notation. In our protocol, **procedures** are input actions that can be invoked at any time (e.g., in a higher level protocol) by the set's owner, the designated reader/writer process for which this instance of the protocol is providing virtual stable storage; **functions** are private methods; and **upon** clauses specify how processes handle external events (i.e., messages, the global startup event, and the transition from DOWN to RECOVERING).

We use **guards** to prevent actions from being activated under certain conditions. If the **guard** of a message handler is not satisfied, the message is dropped, and the action is not executed. If the **guard** of a **procedure** is not satisfied, that **procedure** is inactive and fails upon invocation.

There is a set of $n$ processes, $\Pi$, and every process has a unique ID in $\{1, 2, ..., n\}$. Each process maintains a *crash vector* of length $n$, with one entry for each process in system. Entry $i$ in this vector (called $i$'s *crash ID*) tracks the latest incarnation of process $i$. When a process recovers, it gets a new value from its local, monotonic clock and updates its crash ID in its own vector. When the recovery procedure ends, the process becomes OPERATIONAL and signals this through the *op* flag. A process's crash vector is updated whenever a process learns about a newer incarnation of another process. Crash vectors are partially ordered, and a join operation, denoted $\sqcup$, is defined over vectors, where $(v_1 \sqcup v_2)[i] = \max(v_1[i],\ v_2[i])$.

The crash vector has two purposes: (1) to match requests with their replies (e.g., to ignore old replies), and (2) to detect whether a process has crashed and recovered. Initially, each process's crash vector is $[\bot, \ldots, \bot]$, where $\bot$ is some value smaller than any value ever returned by any clock, and thus smaller than the crash ID of any process that has ever crashed and begun recovery.

The single main function of our algorithm, ACQUIRE-QUORUM, handles both writing values and recovering. ACQUIRE-QUORUM ensures the persistence of both the process's current crash vector (in particular, the process's own crash ID in the vector) as well as the value to be written ($\bot$, a unique value which every process's local set contains, in the case of recovery). Additionally, it updates the process's local set to contain all previously written values. This guarantees that upon recovery, if the set owner calls READ, it will get those values.

ACQUIRE-QUORUM provides these guarantees by collecting replies from a quorum of processes and ensuring that those replies are *consistent* per Section 5.3. It uses crash vectors to detect when any process that previously replied *could have crashed* and thus "forgotten" about the written value and the process's crash ID. The DISCARD-OLD-REPLIES function detects and removes these replies from the reply set and then resends the original message. While recovering, processes do not send replies for WRITEs or other processes' recoveries; all ACQUIRE-REP messages are sent by OPERATIONAL processes.

### 5.3 Protocol Correctness

Prior to discussing the correctness of Algorithm 1, we first define basic terms.

**Definition 1.** A quorum, $Q$, is a set of processes such that:

$$Q \in \mathcal{Q} = \{Q \mid Q \in 2^{\Pi} \text{ and } |Q| > |\Pi|/2\}$$

Note that $\forall Q_1, Q_2 \in \mathcal{Q}$, $Q_1 \cap Q_2 \neq \varnothing$ (Quorum Intersection).

**Definition 2.** We say that property $X$ is *stable* if the following two guarantees hold:

1. If a process, $p$, has property $X$, then as long as $p$ does not crash, $p$ will still have property $X$.
2. If a process, $p$, has property $X$ and $p$ sends an ACQUIRE or ACQUIRE-REP message to $p'$, then upon receiving the message, $p'$ will have property $X$.

Note that a process having a crash vector, $v$, that is greater than or equal to some value is a stable property. Also, a process having some element in its local set, $S$, is a stable property.

**Definition 3.** If a process, $p$, has some stable property, $X$, we say $p$ *knows* $X$. If a message, $m$, from some process, $p$, indicates that at the time $p$ sent $m$, $p$ knew $X$, then we say $m$ *shows* $X$ (for $p$).

**Definition 4.** We say that a quorum $Q$ *knows* some stable property $X$ if, for all processes $p \in Q$, one of the following holds: (1) $p$ is DOWN, (2) $p$ is OPERATIONAL, and $p$ knows $X$, or (3) $p$ is RECOVERING and is guaranteed to know $X$ upon finishing recovery.

**Definition 5.** A set of ACQUIRE-REP messages, $R$, is *consistent* if:

$$\forall s_1, s_2 \in R : s_1.v[s_2.f] \leqslant s_2.v[s_2.f]$$

**Definition 6.** We say that a consistent set of ACQUIRE-REP messages constitutes a *quorum promise* showing stable property $X$ if the set of senders of those messages is a quorum, and each message shows $X$.

The DISCARD-OLD-REPLIES function (line 32) guarantees the consistency of the reply set by discarding any inconsistent messages; it also guarantees that there is at most one message from each process in the reply set. Therefore, the termination of ACQUIRE-QUORUM (line 13) means that the process has received a quorum promise showing that *val* was written and that every participant had a crash vector greater than or equal to its own vector *when it sent the* ACQUIRE *messages*. This implies that whenever a process finishes recovery and sets its *op* flag to *true*, it must have received a quorum promise showing that the participants in its recovery had that process's latest crash ID in their crash vectors.

**Definition 7.** We say that a process *participates* in a quorum promise for $X$ when it sends an ACQUIRE-REP message that *will eventually* belong to a quorum promise that some other process receives.

Unlike having a stable property, that a process *participated* in a quorum promise holds across failures and recoveries. That is, we say that a process, *not* a specific incarnation of that process, participated in a quorum promise. Also note that only OPERATIONAL processes ever participate in a quorum promise, guaranteed by the **guard** on the ACQUIRE message handler.

**Theorem 1** (Persistence of Quorum Knowledge)**.** *If at time $t$, some quorum, $Q$, knows stable property $X$, then for all times $t' \geqslant t$, $Q$ knows $X$.*

*Proof.* We prove by (strong) induction on $t'$ that the following invariant, $I$, holds for all $t' \geqslant t$. For all $p$ in $Q$: (1) $p$ is OPERATIONAL and knows $X$, (2) $p$ is RECOVERING, or (3) $p$ is DOWN. In the base case at time $t$, $Q$ knows $X$ by assumption, so $I$ holds.

Now, assuming $I$ holds at all times $t' - 1 \geqslant t$, we show that $I$ holds at time $t'$. Because $X$ is stable, the only step any process, $p$, in $Q$ could take to falsify $I$ is finishing recovery. Either recovery began after time $t$, or at or before time $t$. If the latter, then because $Q$ knew $X$, $p$ is guaranteed to know $X$ now that it has finished recovering. If this recovery began after time $t$, then $p$ must have received some set of ACQUIRE-REP messages from a quorum, all of which were sent after time $t$. By quorum intersection, one of these messages must have come from some process in $Q$. Call this process $q$. Since $q$'s ACQUIRE-REP message, $m$, was sent after time $t$ and before $t'$, by the induction hypothesis, $q$ must have known $X$ when it sent $m$. Since $X$ is a stable property, $p$ now knows $X$ upon finishing recovery.

Since $I$ holds for all times $t' \geqslant t$, this implies the theorem. □

**Theorem 2** (Acquisition of Quorum Knowledge)**.** *If process $p$ receives a quorum promise showing stable property $X$ from quorum $Q$, then $Q$ knows $X$.*

*Proof.* We again prove this theorem by (strong) induction, showing that the following invariant, $I$, holds for all times, $t$:

1. If $p$ receives a quorum promise showing stable property $X$ from quorum $Q$, then $Q$ knows $X$.
2. If $p$ ever participated in a quorum promise for $X$ at or before time $t$, and $p$ is OPERATIONAL, then $p$ knows $X$.

In the base case at $t = 0$, $I$ holds vacuously since no process could have yet received or participated in a quorum promise. Now, assuming $I$ holds at time $t - 1 \geqslant 0$, we will show that $I$ holds at time $t$.

We prove our invariant in two parts. First, let's dispatch with property 1 of $I$. If $p$ has received a quorum promise, $R$, from quorum $Q$ showing $X$, then because $R$ is consistent, we know that *at the time they participated in* $R$ no process in $Q$ had participated in the recovery[3] of any later incarnation of any other process in $Q$ than the one that participated in $R$. If they had, then by the induction hypothesis (which we can apply as their participation happened before time $t$), we would know that such a process would have known the recovered process's new crash ID in the crash vector when it participated in $R$, and $R$ would not have been consistent.

Given that fact, we will use a secondary induction to show that for all times, $t'$, all of the processes in $Q$ either: (1) haven't yet participated $R$, (2) are DOWN, (3) are RECOVERING, or (4) are OPERATIONAL and know $X$.

In the base case at $t' = 0$, no process in $Q$ has yet participated in $R$. Now, for the inductive step, note that the only step any process, $q$, could take that would falsify our invariant is transitioning from RECOVERING to OPERATIONAL after having participated in $R$. We know that if $q$ finished recovering, it must have received a quorum promise showing that the senders knew its new ID in the crash vector. By quorum intersection, at least one of these came from some process in $Q$; call this process $r$. We already know $r$ couldn't have participated in $q$'s recovery before participating in $R$. So by the induction hypothesis, $r$ knew $X$ at the time it participated in $q$'s recovery. And since $X$ is stable, $q$ knows $X$, completing this secondary induction.

Finally, we know that since $p$ has received $R$, at time $t$ all of the processes in $Q$ have already participated in $R$, so all of the processes in $Q$ are either DOWN, RECOVERING (and guaranteed to know $X$ upon recovery), or are OPERATIONAL and know $X$. Therefore, $Q$ knows $X$, and this completes the proof that property 1 of $I$ holds at time $t$.

Now, let's deal with property 2 of $I$. Suppose, for the sake of contradiction, that at or before time $t$, $p$ participated in quorum promise $R$ showing $X$ (i.e. sent some message showing $X$ that is, or will be, in a quorum promise) received by $q$. Further suppose that $p$ is OPERATIONAL, and $p$ doesn't know $X$. Let $Q$ be the the set of participants in $R$ (i.e. those processes which already have or will at some point participate in the quorum promise, $R$).

Since $X$ is a stable property, $p$ must have crashed and recovered since participating in $R$. Consider $p$'s most recent recovery, and let the quorum promise it received showing that the senders knew $p$'s new crash ID in the crash vector (or a greater one) be $R'$. Let the set of participants in $R'$ be $Q'$. By quorum intersection, there exists some process in $Q \cap Q'$. Consider one such process, $r$.

We know that $r$ couldn't have already participated in $R$ when it participated in $R'$; otherwise by induction, when $r$ participated in $R'$, it would have known $X$, so our incarnation of $p$ at time $t$ would know $X$. Therefore, $r$ participated in $R'$ before participating in $R$. $r$ couldn't have participated in $R$ before time $t$, however; otherwise, by property 2 of $I$, it would have known $p$'s latest crash ID when participating in $R$, violating the consistency of $R$.

However, we know that $p$ has received a quorum promise for its new crash ID at or before time $t$. So by property 1 of $I$, which we have already shown holds at time $t$, $Q'$ knows $p$'s new crash ID. And, by Theorem 1, if $Q'$ knows $p$'s new crash ID at some time less than or equal to $t$, then for all times greater than or equal to $t$, $Q'$ knows $p$'s new crash ID. In particular, this means that for all times greater than or equal to time $t$, if $r$ is up, it knows $p$'s new crash ID. Therefore, $r$ can't participate in $R$ at or after time $t$, contradicting the fact that $r$ participates in $R$ and completing the proof that property 2 holds at time $t$. □

Finally, as stated above, if WRITE terminates at time $t$, we know that the process received promise from a quorum, $Q$, showing that *val* was written. So by Theorem 2, we know this means $Q$ knows *val*, and by Theorem 1, we know that $Q$ will continue to know *val*. If

[3] That is, participated in the quorum promise needed by a recovering process, showing that the senders knew the recovering process's new ID (or a greater one) in the crash vector.

that process ever crashes after time $t$ and subsequently recovers, we know that upon finishing recovery, it must know *val* since it must have received an ACQUIRE-REPLY from some process in $Q$ which knew *val*. Therefore, once a WRITE terminates, any subsequent READ will at least return the written value, showing the safety property of virtual stable storage.

**Termination** It would be nice to guarantee the termination of WRITE and recovery in all circumstances. However, this is clearly impossible. In the DCR model, processes can experience "amnesia." Consider some process, $p$, trying to recover or write a value. If every time any other process receives a message from $p$, it replies, and then crashes and recovers (before any other process receives a message from $p$), then it is obviously impossible for $p$ to build up distributed knowledge, much less a quorum promise.

Our single reader, single writer set protocol does, however, guarantee termination of both WRITE and recovery if there is some quorum of processes, all of which remain OPERATIONAL for a sufficient period of time (and the writing or recovering process itself does not crash). This is easy to see since a writing or recovering process will eventually get an ACQUIRE-REP from each of these OPERATIONAL processes, and those replies must be consistent (since no process could have a crash ID for another process in its vector higher than the crash ID of the latest incarnation of that process).

Furthermore we note that even though our protocol is always safe, if ever a majority of processes is DOWN any given time, then no process can ever receive replies from a quorum again, so no process will ever be able to recover or write again.[4]

# 6 Related work

Early work on the Crash-Recovery failure model proposed protocols to solve the consensus problem [3, 14]. These papers assume stable storage, and do not focus on state-loss failures, either assuming that processes do not lose state on crash [14] or that every state transition is persisted to stable storage [3]. (Others persist only *critical* state to stable storage [5].) As a result, the main focus of this work is *omission failures*, i.e., the messages that processes do not receive while down.

Aguilera et al. [1] were the first to study consensus in a Crash-Recovery failure model without stable storage. The authors proved that consensus is impossible to solve in this model unless the number of *always-up* processes exceeds the upper bound on processes that are either eventually-always-down or unstable. Their failure model, however, differs significantly from ours: it does not make a distinction between recovering and operational processes. As a result, it cannot allow processes to recover their state, and hence at least one process must remain always up. Instead, we show that it is possible to provide stable storage in the Diskless Crash-Recovery model as long as there is a quorum of processes that remain operational for a sufficient time. This allows any process to crash and fail at some point during the execution of the protocol, which is expected to happen in long-lived systems.

An interesting trade-off between stable storage and a majority of correct process is discussed by Martin et al. [12]. They consider a particular problem in the DCR model, eventual leader election, that has weaker requirements than emulating stable storage in this model. In particular, this problem can be solved even when processes do not always recover the state that they lost.

# 7 Conclusion

This paper defines the Diskless Crash-Recovery model (DCR), which supports long-lived distributed services where nodes may crash and rejoin the system without their volatile state. Building a correct recovery protocol for this environment is surprisingly subtle; to our knowledge, it has never been addressed in general. All previous protocols for the specific case of state machine replication in this model violate safety under certain failure scenarios. We present the first general procedure for transforming Crash-Stop or Crash-Recovery algorithms that rely on stable storage to the DCR model. Our algorithm, which uses a crash vector to achieve global knowledge about recovery progress, remains correct in all cases and terminates as long as a majority of nodes remain operational.

[4] In fact, if there is ever a majority of processes that are DOWN or RECOVERING (where there does not exist a set of messages currently in the network that will allow any of them to recover), then no process will ever be able to recover or write again.

## References


[1] M. K. Aguilera, M. Kawazoe, A. Wei, and C. S. Toueg. Failure Detection and Consensus in the Crash-Recovery Model. In *Proc. of DISC*, 1998.

[2] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proc. of PODC*, 2007.

[3] D. Dolev, R. Friedman, I. Keidar, and D. Malkhi. Failure detectors in omission failure environments. Technical report, Department of Computer Science, Cornell University, 1997.

[4] M. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 1990.

[5] M. Hurfin, A. Mostéfaoui, and M. Raynal. Consensus in Asynchronous Systems Where Processes Can Crash and Recover. In *Proc. of the 17th Symposium on Reliable Distributed Systems, SRDS*, 1998.

[6] J. Kończak, N. Santos, T. Żurkowski, P. T. Wojciechowski, and A. Schiper. JPaxos: State machine replication based on the Paxos protocol. Technical report, 2011.

[7] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 1978.

[8] L. Lamport. On interprocess communication, 1986.

[9] L. Lamport. Paxos made simple. *ACM SIGACT News 32*, 2001.

[10] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical report, 2012.

[11] N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. Technical report, MIT Laboratory for Computer Science, Cambridge, Mass., 1988.

[12] C. Martin, M. Larrea, and E. Jimenez. Implementing the Omega Failure Detector in the Crash-Recovery Failure Model. *J. Comput. Syst. Sci.*, 2009.

[13] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. of PODC*, 1988.

[14] R. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the Crash Recover Model. Technical report, Département d'Informatique, École Polytechnique Fédérale, Lausanne, Switzerland, 1997.

[15] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 1990.

# A Description of Safety Violations in Existing SMR Solutions

As mentioned in Section 4, three prior systems have proposed protocols for *state machine replication* in the Diskless Crash-Recovery model. This appendix shows safety violating traces for these protocols.

## A.1 Viewstamp Replication Revisited

Viewstamped Replication Revisited [10] is a recent version of the classic VR protocol [13] that specifically targets the DCR model.

**Basic VR Protocol.** VR is a leader-based algorithm; the system moves through a series of views in which one node is designated as the leader. That node is responsible for assigning an order to operations. When the leader receives a request from a client, it sends a PREPARE message containing the request and sequence number to the other replicas. Upon receiving a PREPARE, a replica verifies that it is in the same view as the leader and then records the request in its log and responds to the leader with a PREPARE-OK. Once the leader receives PREPARE-OK messages from a majority of replicas, it executes the request, responds to the client, and sends a COMMIT message to the other replicas. The other replicas then execute the request.

VR uses a view change protocol to mask failures of the leader. If a replica suspects the leader of having failed, it notifies the other replicas. These replicas stop processing requests, increment their view number, stop processing requests in the old view, and send a DO-VIEW-CHANGE message to the leader of the new view. This request includes the log of operations previously executed or prepared by that replica. Once the new leader receives DO-VIEW-CHANGE messages from a majority of replicas, it selects the longest log, and sends a START-VIEW message to notify the other replicas that they can resume normal operation. This protocol ensures that all successfully completed operations persist across view changes: each such operation must have completed at a majority of replicas, and a majority of replicas send logs to their leader, so the new leader will have learned about that operation from at least one replica.

As described in Section 4, a key invariant in VR is that once a node has sent a DO-VIEW-CHANGE message to the leader of a view, it must never accept new operations in prior views – otherwise these operations could be lost after the view change. A recovery protocol must ensure that this property continues to hold even if replicas crash and recover.

**Diskless Recovery.** The original version of VR used stable storage for logging view numbers during view changes (and only for this purpose). VR Revisited eliminates this use of stable storage, attempting to emulate it with a write to a quorum.

To achieve this goal, VR Revisited uses two additions to the protocol, described in Section 4. First, it introduces a recovery protocol, where a recovering node contacts all other replicas and waits for a response from a quorum of replicas with matching view number. Second, it adds another phase to the view change protocol: nodes stop processing requests when they notice the need for a view change and send a START-VIEW-CHANGE message to all other replicas; they only send the DO-VIEW-CHANGE message upon receiving START-VIEW-CHANGE messages from a quorum of replicas.

**Failure Trace.** Simply replacing a log to stable storage with a write to a quorum is not sufficient to ensure correct recovery. In Figure 1, we show a trace with 3 processes, in which a new leader (NL) mistakenly overwrites the decision of a previous leader (OL).

- Initially, NL suspects OL of failing, and sends a START-VIEW-CHANGE message to node 1 to switch to *view 1*.
- NL crashes immediately after sending this message – before node 1 receives it – then immediately initiates recovery. It sends RECOVERY messages and receives RECOVERY-RESPONSE messages from OL and 1, both of which are in view 0 – so NL recovers in view 0.
- Node 1 receives the START-VIEW-CHANGE message sent by the previous incarnation of NL
- Node 1 sends a START-VIEW-CHANGE message to NL for view 1. Because node 1 has a quorum of START-VIEW-CHANGE messages for view 1 (its own and the one from NL), it also sends a DO-VIEW-CHANGE message to NL. Both messages are delayed by the network.
- Node 1 crashes and immediately recovers, sending RECOVERY messages to and receiving responses from OL and NL – both of which are in view 0.
- NL receives START-VIEW-CHANGE and DO-VIEW-CHANGE messages from node 1. It has a quorum of START-VIEW-CHANGE messages, so it sends a DO-VIEW-CHANGE message for view 1. This is enough for it to complete the view change.

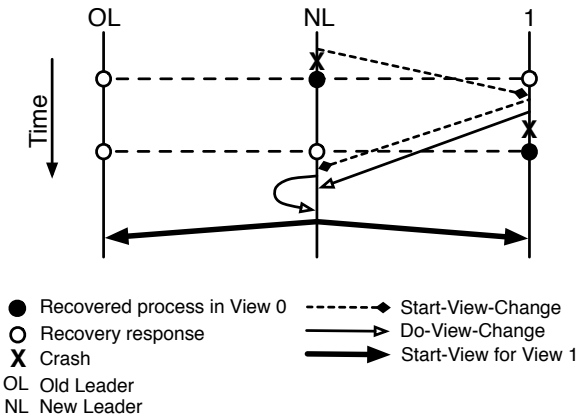


Figure 1: Trace showing safety violation in Viewstamped Replication Revisited [10]

It sends a START-VIEW message.

- Until the START-VIEW message is received, nodes OL and 1 are still in view 0 and do not believe a view change is in progress. Thus, they can commit new operations, which the new leader NL will not know about, leaving the system in an inconsistent state..

In this trace, messages are reordered inside the network. In particular, messages are reordered across failures, i.e., messages from a prior incarnation of a process are sometimes delivered *after* messages from a later one. A network with FIFO communication channels would not allow the violation described above: recovering nodes will receive a reply to their recovery message only after their previous messages (e.g., START-VIEW-CHANGE message) have been received. However, message reordering is *not required* for this failure case: we have found a a trace with 7 nodes that leads to the same behavior, even with FIFO communication channels, as shown in Figure 2.

## A.2 Paxos Made Live

Paxos Made Live is Google's production implementation of a Paxos [2]. It is based on the well-known Multi-Paxos optimization which chains together multiple instances of Paxos [9] and is effectively equivalent to VR. This system primarily uses stable storage to support crash recovery, but because disks can become corrupted or otherwise fail, the authors propose a version that allows recovery without disks.

**Recovery Protocol.** On recovery, a process first uses a *catch-up* mechanism to bring itself up-to-date. The specific mechanism it uses is not described, but presumably it is an application-level state transfer from a quorum of correct processes as in VR. In order to ensure consistency, the recovering process is not allowed to participate in the protocol until it observes a completed instance of successful consensus after its recovery, i.e., until it learns that at least a quorum have agreed on a value for a new consensus instance. This mechamism suffers from a similar problem to the one in VR. Although it protects against losing ACKNOWLEDGEMENTs (i.e., PREPARE-OK messages in VR), it does not protect against losing PROMISEs made to potential new leaders (i.e., VR's DO-VIEW-CHANGE messages).

**Failure Trace.** Figure 3 shows a trace with 5 processes that leads to a new leader mistakenly deciding a new value for a prior successful instance of consensus, overwriting the decision of the previous leader:

- Initially, node OL is the leader.
- Node NL suspects the leader of having failed, so sends a PROPOSE message proposing itself as the next leader.

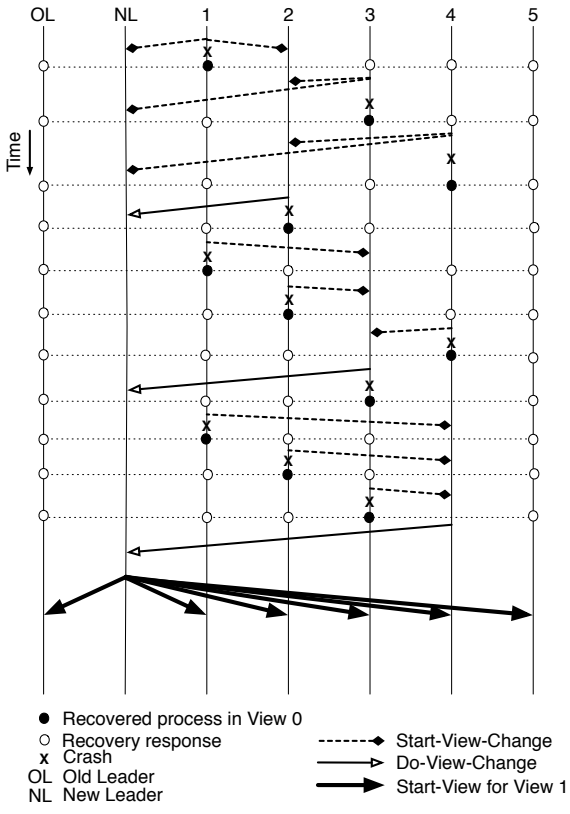


Figure 2: Trace showing safety violation in Viewstamped Replication Revisited [10], assuming FIFO channels

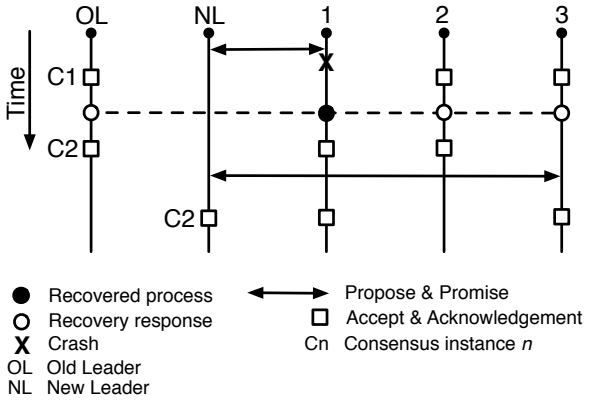


Figure 3: Trace showing safety violation in Paxos Made Live [2]

- Node 1 receives NL's proposal, and sends a PROMISE to NL, promising that it will not accept any further operations from OL.
- Node 1 crashes and immediately recovers.
- OL selects a new value for the next instance, C1, and sends ACCEPT and receives ACKNOWLEDGEMENT messages from nodes 2 and 3. The operation is committed, and node 1 completes recovery because it has now observed an instance of consensus.
- OL selects a value for instance C2, and sends ACCEPTs and receives ACKNOWLEDGEMENTs from nodes 1 and 3.
- Node 3 then receives NL's PROPOSE and sends NL a PROMISE.
- NL has now received a quorum of PROMISE messages: from itself, node 3, and the previous incarnation of node 1. None of these observed consensus instance C2, so NL can now start instance 2 of consensus and overwrite the previous value with its own ACCEPT messages.

### A.3 JPaxos

JPaxos [6] is a state machine replication protocol based on Paxos. It is a hybrid between Multi-Paxos and VR, replacing *promises* with *views*. The protocol is presented in several deployment options, including one that does not use stable storage, i.e., supports the Diskless Crash-Recovery model.[5]

**Recovery Protocol.** JPaxos provides a protocol for the DCR model called *epoch-based recovery*. This protocol is similar in spirit to our use of crash vectors: it attempts to maintain a vector clock of how many times each node has crashed and recovered. Each node updates this vector when it sends a recovery response to another node, and includes a copy of its vector in the PROPOSE-OK messages that it uses to commit to a new view (i.e., the equivalent of the DO-VIEW-CHANGE message). It then uses these vectors to discard all PROPOSE-OK messages that do not come from a consistent quorum.

JPaxos's epoch-based recovery protocol has one critical difference from ours. It treats recovery as complete once it has received RECOVERY-ANSWER messages from a quorum of replicas. In contrast, our protocol requires recovery responses from a *consistent* quorum of replicas. That is, it discards recovery responses from the quorum if it learned that the replica that sent the response itself crashed and recovered.

This subtle change has important implications for correctness. Without this, JPaxos does not correctly handle recovery of nodes when *the nodes they are recovering from* can themselves crash and recover. When this happens, JPaxos can allow a node to recover in a way that simultaneously 1) causes the node to lose the promise that it made in a previous PREPARE-OK, yet 2) does not allow other nodes to detect that the promise might have been lost. We give such an example below.

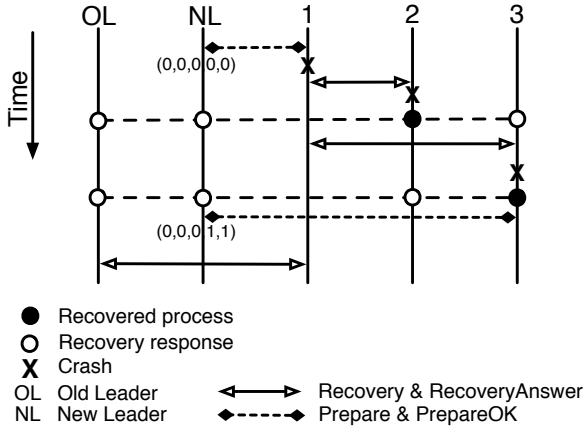


Figure 4: Trace showing safety violation in JPaxos

**Failure Trace.** Figure 4 shows a trace in which a process (node 1 in the figure) commits to a new view via a PREPARE-OK message, then crashes and recovers. When it recovers, it recovers in the previous view without having learned of its promise; however, the new leader (NL) does not learn about node 1's crash and recovery, and therefore does not discard its PREPARE-OK message.

[5]The protocol is actually presented as using stable storage on process initialization, but only to store a monotonic value on write. The monotonic counter available in our DCR model fills this need.

- The system starts in a view where OL is the leader, and all nodes have epoch vector $(0,0,0,0,0)$.
- Node NL suspects OL of having failed, so sends out a PREPARE message proposing itself as the leader of the next view.
- Node 1 receives NL's PREPARE message and sends a PREPARE-OK.
- Node 1 crashes and immediately recovers. It sends a RECOVERY message to node 2, and receives a RECOVERY-ANSWER. Node 2's epoch vector is now $(0,0,1,0,0)$
- Node 2 crashes and immediately recovers. It sends a RECOVERY message and receives RECOVERY-ANSWERs from nodes OL, NL, and 3. All of these have epoch vector $(0,0,0,1,0)$, so 2 now has this vector as well – in other words, it has lost its knowledge that 1 crashed.
- Node 1 sends a RECOVERY message to node 3, and receives a reply. Node 3's epoch vector is now $(0,0,1,1,0)$.
- Node 3 crashes and immediately recovers, communicating with nodes OL, NL, and 2 during recovery. After recovery, its epoch vector is $(0,0,0,1,1)$.
- NL sends a PREPARE message to 3, and receives a PREPARE-OK responses. It now has a quorum of PREPARE-OK responses from itself, 1, and 3, so it can start a new view.
- Node 1 sends a RECOVERY message to node OL, and receives a response. It is now fully recovered in the original view.
- OL can now commit operations (via the quorum of itself, 1, and 2) which will not appear in NL's new view.

Our protocol avoids this problem by checking for a *consistent* quorum on recovery. When node 1 receives a recovery response from OL, that response will have crash vector $(0,0,0,1,1)$ – and so node 1 will discard the earlier recovery responses it received from 2 and 3. It does so because it has learned that those nodes have crashed and recovered, and therefore their updates to the crash vector may not be stable.