Recovering Shared Objects Without Stable Storage[*][†]

Ellis Michael[1], Dan R. K. Ports[2], Naveen Kr. Sharma[3], and Adriana Szekeres[4]

1 **University of Washington, Seattle, USA**
`emichael@cs.washington.edu`

2 **University of Washington, Seattle, USA**
`drkp@cs.washington.edu`

3 **University of Washington, Seattle, USA**
`naveenks@cs.washington.edu`

4 **University of Washington, Seattle, USA**
`aaasz@cs.washington.edu`

Abstract

This paper considers the problem of building fault-tolerant shared objects when processes can crash and recover but lose their persistent state on recovery. This Diskless Crash-Recovery (DCR) model matches the way many long-lived systems are built. We show that it presents new challenges, as operations that are recorded at a quorum may not persist after some of the processes in that quorum crash and then recover.

To address this problem, we introduce the notion of *crash-consistent quorums*, where no recoveries happen during the quorum responses. We show that relying on crash-consistent quorums enables a recovery procedure that can recover all operations that successfully finished. Crash-consistent quorums can be easily identified using a mechanism we term the *crash vector*, which tracks the causal relationship between crashes, recoveries, and other operations.

We apply crash-consistent quorums and crash vectors to build two storage primitives. We give a new algorithm for multi-writer, multi-reader atomic registers in the DCR model that guarantees safety under all conditions and termination under a natural condition. It improves on the best prior protocol for this problem by requiring fewer rounds, fewer nodes to participate in the quorum, and a less restrictive liveness condition. We also present a more efficient single-writer, single-reader atomic set – a *virtual stable storage* abstraction. It can be used to lift any existing algorithm from the traditional Crash-Recovery model to the DCR model. We examine a specific application, state machine replication, and show that existing diskless protocols can violate their correctness guarantees, while ours offers a general and correct solution.

1998 ACM Subject Classification E.1 Distributed Data Structures, H.3.4 Distributed Systems

Keywords and phrases asynchronous system, fault-tolerance, crash-recovery, R/W register, state machine replication

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.36

[*] An extended version of this paper is available as a technical report [27].
[†] This material is based upon work supported by the National Science Foundation under award CNS-1615102 and a Graduate Research Fellowship, and by gifts from Google and VMware.

1 Introduction

Today's distributed systems are key pieces of infrastructure that must remain available even though the servers that implement them are constantly failing. These systems are long-lived and must be able to tolerate nodes crashing and rejoining the system. In particular, nodes must be able to rejoin the system even after losing their disk state, a real concern for large-scale data centers where hard drive failures are a regular occurrence [7].

This paper addresses the problem of how to build recoverable shared objects even when processes lose their entire state. We consider the *Diskless Crash-Recovery* model: each process in the system may go down at any time; upon recovery, it loses all state it had before the crash except for its identity. However, processes can run a *recovery protocol* to reconstruct their state before deeming themselves operational again. This model matches the way that many distributed systems are built in practice.

The Diskless Crash-Recovery model (DCR) is more challenging than the traditional Crash-Stop model (CS) or the Crash-Recovery with Stable Storage model (CRSS). The main challenge is that an invariant that holds at one process may not hold on that process's next incarnation after recovery. This leads to the problem of *unstable quorums*: it is possible for a majority of processes to acknowledge a write operation, and yet processes can still subsequently lose that knowledge after crash and recovery.

We provide a general mechanism for building recoverable shared objects in the DCR model. We show that an operation can be made recoverable once it is stored by a *crash-consistent quorum*, which we informally define as one where no recoveries happen during the quorum responses. Crash-consistent quorums can be efficiently identified using a mechanism called the *crash vector*: a vector, maintained by each process, that tracks the latest known incarnation of each process. By including crash vectors in protocol messages, processes can identify the causal relationship between crash recoveries and other operations. This makes it possible to discard responses that are not part of a crash-consistent quorum. We show that this is sufficient to make storage mechanisms recoverable.

The crash-consistent quorum approach is a general strategy for making storage primitives recoverable. We give two concrete examples in this paper, both of which are always safe and guarantee liveness during periods of stability; other storage primitives are also possible:

- First, we build a *multi-writer, multi-reader atomic register* by extending the well-known ABD protocol [3] with crash vectors. This improves on the best prior protocol by Konwar et al. [17], $RADON_R^{(S)}$, for this problem: it requires fewer rounds (2 rather than 3), requires fewer nodes to participate in the protocol (a simple majority vs $3/4$), and has a less restrictive liveness condition.
- Second, we construct a *single-writer, single-reader atomic set*, which has weaker semantics yet permits a more efficient implementation, requiring only a single round of communication for writes. We refer to this algorithm as *virtual stable storage*, as it offers consistency semantics similar to a local disk. We show that the virtual stable storage protocol can be used to transform any protocol that operates in the traditional CS or CRSS models to one that operates in DCR.

We discuss the application of this work to state machine replication, a widely used distributed system technique. Recovering from disk failures is an important concern in practice, and recent replication protocols attempt to support recovery after complete loss of state. Surprisingly, we find that each of the three such protocols [7, 16, 22] can lose data. We identify a general problem: while these protocols go to great lengths to ensure that a recovering replica reconstructs the set of operations it previously processed, they fail to recover critical *promises* the replica has previously made, e.g., to elect a new leader. This is due to the fact that these protocols rely on *unstable quorums* to persist these promises. This causes nodes to break important invariants upon recovery, causing the system to violate safety properties. Our approach provides a correct, general, and efficient solution.

To summarize, this paper makes the following contributions:

- It formalizes a *Diskless Crash-Recovery (DCR)* failure model in a way that captures the challenges of long-lived applications (Section 3).
- It introduces the notion of *crash-consistent quorums* and provides two communication primitives for reading from and writing to crash-consistent quorums (Section 4).
- It presents algorithms built on top of our communications primitives for two different shared objects in the DCR model: an atomic multi-writer, multi-reader register and an atomic single-writer, single-reader set. The former is a general purpose register which demonstrates the generality of our approach, while the latter provides a *virtual stable storage* interface that can be used to port any protocol in the CRSS model to one for the DCR model (Section 5).
- Finally, it examines prior protocols for state machine replication in the DCR failure model and demonstrates flaws in these protocols that lead to violations of safety properties. Our two communication primitives can provide correct solutions (Section 6).

2 Background and Related Work

Static Systems. A static system comprises a fixed, finite set of processes. Fault-tolerant protocols for reliable storage for static systems have been studied extensively in the Crash-Stop (CS) failure model, where processes that fail never rejoin the system, and the Crash-Recovery with Stable Storage model (CRSS). In the latter model, processes recover with the same state after a crash. Consensus and related problems, in particular, have been studied extensively in these settings [8, 12, 30]. In CRSS, a crashed and recovered node is no different than one which was temporarily unavailable; asynchronous algorithms that tolerate lossy networks are inherently robust to these types of failures [8].

Prior work on fault-tolerant shared objects and consensus without stable storage generally requires some subset of the processes to never fail [2, 11]. Aguilera et al. [2] showed an impossibility result for a crash-recovery model: even with certain synchrony assumptions, consensus cannot be solved *without at least one process that never crashes*. The main differentiator between that work and this paper is that in their model, the states of processes were binary – either "up" or "down." We overcome this limitation by adding an extra "recovering" state. As long as the number of processes which are "down" or "recovering" at any given time is bounded, certain problems can be solved even *without processes that never fail*.

Recently, Konwar et al. [17] presented a set of algorithms for implementing an atomic multi-writer, multi-reader (MWMR) register in a model similar to ours. We generalize and improve on this work using new primitives for crash-consistent quorums. Our techniques are applicable to other forms of shared objects as well, and our MWMR register is more efficient: it requires one fewer phase and a simple majority quorum (vs $3/4$).

Several recent practical state machine replication systems [7, 16, 22] incorporate ad hoc recovery mechanisms for nodes to recover from total disk loss. The common intuition behind these approaches is that a write to disk can be replaced with a write to a quorum of other nodes, recovering after a failure by performing a quorum read. However, we show that these protocols are not correct; they can lose data in certain failure scenarios. A more recent design, Replacement [13], provides a mechanism for replacing failed processes. Like our work and the epoch vectors in JPaxos [16], it draws on concepts like version vectors [31] and vector clocks [9] to determine the causal dependencies between replacements and other operations. We build on these techniques to provide generic communication primitives in DCR.

Dynamic Systems. In a dynamic setting, processes may leave or join the system at will. Although we consider a static system, DCR may be viewed as a dynamic system with a finite concurrency level [26], i.e, where there is a finite bound on the maximum number of processes that are simultaneously active, over all runs. Here, a recovering process without state is equivalent to a newly joined process.

Many dynamic systems implement *reconfiguration* protocols [1,10,21–24,33]. Reconfiguration allows one to change the set of members allowed to participate in the computation. This process allows both adding new processes and removing processes from the system. Reconfiguration is a more general problem than recovery: it can be used to handle disk failure by introducing a recovering node as a new member and removing its previous incarnation. However, general reconfiguration protocols are a blunt instrument, as they must be able to handle completely changing the membership to a disjoint set of processes . As a result, these protocols are costly. Most use consensus to agree on the order of reconfigurations, which delays the processing of concurrent operations [28]. DynaStore [1] is the first proposal which does not require consensus, but reconfigurations can still delay R/W operations [28]. Smart-Merge [14] improves on DynaStore by offering a more expressive reconfiguration interface. Recovery is a special case of reconfiguration, where each recovering process replaces, and has the same identity as, a previously crashed process. As a result, it permits more efficient solutions.

Other protocols implement shared registers and other storage primitives in churn-prone systems [4–6,15]. In these systems, processes are constantly joining and leaving the system, but at a bounded rate. These protocols remain safe only when churn remains within the specified bound, in contrast to our work which is always safe. Most of these protocols also require synchrony assumptions for correctness. However, under these assumptions they are able to provide liveness guarantees even during constant churn.

3 System Model

We begin by defining our failure model: *Diskless Crash-Recovery* (DCR), a variant of the classic Crash-Recovery model where processes lose their entire state upon crashing.

We consider an asynchronous distributed system which consists of a fixed set of n processes, Π. Each process has a unique name (identifier) of some kind; we assume processes are numbered $1, \dots, n$ for simplicity. Each process executes a protocol (formally, it is an I/O automaton [25]) while it is up. An execution of a protocol proceeds in discrete time steps, numbered with $\mathbb{N}$, starting at $t = 0$. At each step, at most one process either processes an input action, processes a message, crashes, or restarts. If it *crashes*, the process stops receiving messages and input actions, loses its state, and is considered DOWN. A process that is DOWN can *restart* and transition back to the UP state. We make the following assumptions about a process that restarts: (1) it knows it is restarting, (2) it knows its unique name and the names of the other processes in the system (i.e., this information survives crashes), and (3) it can obtain an incarnation ID that is distinct from all the ones that it previously obtained. Note that the incarnation ID need only be unique among different incarnations of a specific process, not the entire system. These are reasonable assumptions to make for real-world systems: (1) and (2) are fixed for a given deployment, and (3) can be obtained, for example, from a source of randomness or the local processor clock.

Processes are connected by an asynchronous network. Messages can be duplicated a finite number of times or reordered arbitrarily – but not modified – by the network. We assume that if an incarnation of a process remains UP, sends a message, and an incarnation of the destination process stays UP long enough, that message will eventually be delivered.[1]

The unique incarnation ID makes it possible to distinguish different incarnations of the same process. Without unique incarnation IDs, processes are vulnerable to "replay attacks:"

▶ **Theorem 1.** *Any state reached by a process that has crashed, restarted, and taken steps without receiving an input action or crashing again will always be reachable by that process.*

Proof. Suppose process p has crashed, restarted, and taken some number of steps without crashing or receiving an input action. That is, suppose that after it restarted, p received some sequence of messages, $\mathcal{M}$. Because p is an I/O automaton without access to randomness or unique incarnation IDs, anytime p crashes and restarts, it restarts into the exact same state. Furthermore, if p crashes, restarts, and receives the same sequence of messages, $\mathcal{M}$, having been duplicated by the network, p will always end up in the same state. ◀

A corollary to Theorem 1 is that any protocol in the DCR model without unique incarnation IDs satisfying the safety properties of consensus – or even a simple shared object such as a register – can reach a state from which terminating states are not reachable (i.e., a state of deadlock). If all processes crash and recover before deciding a value or receiving a write, they can always return to this earlier state, so the protocol cannot safely make progress.

For simplicity of exposition, we assume that the incarnation ID increases monotonically. We explain in our technical report [27] how to eliminate this requirement.

A restarting process must recover parts of its state. To do so, it runs a distinct *recovery protocol*. This protocol can communicate with other processes to recover previous state. Once the recovery protocol terminates, the process declares recovery complete 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 when it is running the initial automaton. A protocol in this model should satisfy *recovery termination*: a recovering process eventually becomes OPERATIONAL, as long as it does not crash again in the meantime. This precludes vacuous solutions where recovering process never again participate in the normal protocol.

Using a separate recovery protocol matches the design of existing protocols like Viewstamped Replication [22]. Importantly, the distinction between RECOVERING and OPERATIONAL allows failure bounds in terms of the number of OPERATIONAL processes, e.g., that fewer than half of the processes can be either DOWN or RECOVERING at any moment. This circumvents Aguilera et al.'s impossibility result for consensus [2], which does not make such a distinction (i.e., restarting processes are immediately considered OPERATIONAL).

4 Achieving Crash-Consistent Quorums

Making shared objects recoverable in the DCR model requires a new type of quorum to capture the idea of persistent, recoverable knowledge. A simple quorum does not suffice. We demonstrate the problem through a simple straw-man example, and introduce the concepts of *crash-consistent quorums* and *crash vectors* to solve the problem. We use these to build generic quorum communication and recovery primitives.

[1] This model is equivalent to one in which the network can drop any message a finite number of times, with the added stipulation that processes resend messages until they are acknowledged.

4.1 Unstable Quorums: Intuition

Consider an intentionally simple example: a fault-tolerant *safe* register that supports a single writer and multiple readers. A safe register [19] is the weakest form of register, as the behavior of READ operations is only defined when there are no concurrent WRITEs. We further constrain the problem by allowing the writer to only ever execute one WRITE operation. That is, the only safety requirement is that once the WRITE completes, all subsequent READs that return must return the value written.

In the Crash-Stop model, a trivial quorum protocol suffices: WRITE(val) broadcasts val and waits for acknowledgments from a quorum. Here, we consider majority quorums:

▶ **Definition 2.** A quorum Q is a set of processes such that $Q \in \mathcal{Q} = \{Q : Q \in 2^{\Pi} \wedge |Q| > n/2\}$.

A subsequent READ would then be implemented by reading from a quorum. The quorum intersection property (i.e., $\forall Q_1, Q_2 \in \mathcal{Q} \;\; Q_1 \cap Q_2 \neq \{\}$) guarantees that at least one process will return val for a READ that happens after the WRITE. It is easy to extend this protocol to the CRSS model simply by having each process log val to disk before replying to a WRITE.

Could we use this same quorum protocol in our DCR model, where processes that crash recover without stable storage, by augmenting it with a recovery protocol that satisfies recovery termination? In fact, for this particular protocol, there is *no* recovery protocol that both guarantees the safety requirement and recovery termination – even if there is a majority of processes which are OPERATIONAL at any instant! In order to tolerate the crashes of a minority of processes and satisfy recovery termination, any recovery protocol must be able to proceed after communicating with only a simple majority of processes. However, if a process crashes in the middle of the WRITE procedure – after acknowledging val – it may recover before a majority of processes have received val. No recovery procedure that communicates only with this quorum of processes can cause the process to relearn val.

We term the resulting situation an *unstable quorum*: the WRITE operation received responses from a quorum, and yet by the time it completes there may no longer exist a majority of processes that know val. It is thus possible to form a quorum of processes that either acknowledged val but then lost it during recovery, or never received the write request (delayed by the network). A subsequent READ could fail by reading from such a quorum.

Although this is a simple example, many important systems suffer from precisely this problem of unstable quorums. We show in Section 6 that essentially this scenario can cause three different state machine replication protocols to lose important pieces of state.

4.2 Crash-Consistent Quorums

We can avoid this problem – both for the straw-man problem above and in the general case – by relying not just on simple quorums of responses but *crash-consistent* ones.

Crash Consistency. We informally define a *crash-consistent quorum* to be one where no recoveries of processes in the quorum *happen during* the quorum responses. More precisely:

▶ **Definition 3.** Let $\mathcal{E}$ be the set of all events in an execution. A set of events, $E \subseteq \mathcal{E}$, is *crash-consistent* if $\forall e_1, e_2 \in E$ there is no $e_3 \in \mathcal{E}$ that takes place at a later incarnation of the same process as e_1 such that $e_3 \rightarrow e_2$. Here, $\rightarrow$ represents Lamport's happens-before relation [18].

In Section 4.3, we show how to build recoverable primitives using crash-consistent quorums, in which all quorum replies (i.e. the message send events at a quorum) are crash-consistent.

Crash Vectors. How does a process acquire a crash-consistent quorum of responses? The mechanism that allows us to ensure a crash-consistent quorum is the *crash vector*. This is a vector that contains, for each process, its latest known incarnation ID. Like a version vector, processes attach their crash vector to the relevant protocol messages and use incoming messages to update their crash vector. The crash vector thus tracks the causal relationship between crash recoveries and other operations. When acquiring a quorum on a WRITE operation, we check whether any of the crash vectors are inconsistent with each other, indicating that a recovery may have happened concurrently with one of the responses. We then discard any responses from previous incarnations of the recovering process, ensuring a crash-consistent quorum, and thus avoiding the aforementioned problem.

4.3 Communication Primitives in DCR

We now describe in detail two generic quorum communication primitives, one of which acquires a *crash-consistent quorum*, as well as a generic recovery procedure. These primitives require their users to implement an abstract interface: READ-STATE, which returns a representation of the state of the shared object; UPDATE-STATE, which alters the current state with a specific value; and REBUILD-STATE, which is called during recovery and takes a set of state representations and combines them.

The ACQUIRE-QUORUM primitive writes a value to a crash-consistent quorum and returns the latest state. The READ-QUORUM primitive returns a *fresh* – but possibly inconsistent – snapshot of the state as maintained at a quorum of processes. If ACQUIRE-QUORUM(val) succeeds, then any subsequent READ-QUORUM will return at least one response from a process that *knows* (i.e., has previously updated its state with) val.

The detailed protocol implementing the two primitives and the recovery procedure 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); **functions** are simple methods; and **upon** clauses specify how processes handle external events (i.e., messages, system initialization, and recovery). We use **guards** to prevent actions from being activated under certain conditions. If the **guard** of a message handler or **procedure** is not satisfied, no action is taken, and the message is not consumed (i.e., it remains in the network undelivered).

Each of the n process in Π maintains a *crash vector*, v, with one entry for each process in the system. Entry i in this vector tracks the latest known incarnation ID of process i. During an incarnation, a process numbers its ACQUIRE and READ messages using the local variable c to match messages with replies. When a process recovers, it gets a new value from its local, monotonic clock and updates its incarnation 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 $(v_1 \sqcup v_2)[i] = \max(v_1[i],\ v_2[i])$. Initially, each process's crash vector is $[\bot, \ldots, \bot]$, where $\bot$ is some value smaller than any incarnation ID.

The ACQUIRE-QUORUM function 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 incarnation ID in the vector – as well as the value to be written, val. It provides these guarantees by collecting responses from a quorum of processes and ensuring that those responses are *crash-consistent*. It uses crash vectors to detect when any process that previously replied *could have crashed* and thus could have "forgotten" the written value.

Algorithm 1 Communications primitives.

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

Volatile Local State:
    $v \leftarrow [\bot$ for $i \in [1, \ldots, n]]$   ▷ Crash vector
    $op \leftarrow false$                         ▷ Operational flag
    $R \leftarrow \{\}$                           ▷ Reply set
    $c \leftarrow 0$                              ▷ Message number

1: upon System-Initialize
2:     $op \leftarrow true$
3: end upon
4: upon Recover
5:     $v[i] \leftarrow$ Read-Clock
6:     $\Sigma \leftarrow$ Acquire-Quorum($null$)
7:     Rebuild-State($\Sigma$)
8:     $op \leftarrow true$
9: end upon
10: function Acquire-Quorum($val$)
11:     $R \leftarrow \{\}$
12:     $c \leftarrow c + 1$
13:     $m \leftarrow \langle$acquire$\rangle$
14:     $m.c \leftarrow c$
15:     $m.val \leftarrow val$
16:     for all $j \in [1, \ldots, n]$ do
17:         Send-Message($m$, $j$)
18:     end for
19:     Wait until $|R| > n/2$
20:     return $\{m.s : m \in R\}$
21: end function
22: function Read-Quorum
23:     $R \leftarrow \{\}$
24:     $c \leftarrow c + 1$
25:     $m \leftarrow \langle$read$\rangle$
26:     $m.c \leftarrow c$
27:     for all $j \in [1, \ldots, n]$ do
28:         Send-Message($m$, $j$)
29:     end for
30:     Wait until $|R| > n/2$
31:     return $\{m.s : m \in R\}$
32: end function
33: function Send-Message($m, j$)
34:     $m.f \leftarrow i$                         ▷ Sender
35:     $m.v \leftarrow v$
36:     Send $m$ to process $j$
37: end function
38: upon receiving $\langle$acquire$\rangle$, $m$
39: guard: $op$
40:     $v \leftarrow v \sqcup m.v$
41:     $m' \leftarrow \langle$acquire-rep$\rangle$
42:     if $m.val \neq null$ then
43:         Update-State($m.val$)
44:     end if
45:     $m'.s \leftarrow$ Read-State
46:     $m'.c \leftarrow m.c$
47:     Send-Message($m'$, $m.f$)
48: end upon
49: upon receiving $\langle$acquire-rep$\rangle$, $m$
50: guard: $m.v[i] = v[i] \wedge c = m.c$
51:     $v \leftarrow v \sqcup m.v$
52:     Add $m$ to $R$
    ▷ Discard inconsistent, duplicate replies
53:     while $\exists m' \in R$ where
54:             $m'.v[m'.f] < v[m'.f]$ do
55:         Remove $m'$ from $R$
56:         Resend $\langle$acquire$\rangle$ message to $m'.f$
57:     end while
58:     while $\exists m', m'' \in R$ where
59:             $m'.f = m''.f \wedge m' \neq m''$ do
60:         Remove $m'$ from $R$
61:     end while
62: end upon
63: upon receiving $\langle$read$\rangle$, $m$
64: guard: $op$
65:     $v \leftarrow v \sqcup m.v$
66:     $m' \leftarrow \langle$read-rep$\rangle$
67:     $m'.s \leftarrow$ Read-State
68:     $m'.c \leftarrow m.c$
69:     Send-Message($m'$, $m.f$)
70: end upon
71: upon receiving $\langle$read-rep$\rangle$, $m$
72: guard: $m.v[i] = v[i] \wedge c = m.c$
73:     $v \leftarrow v \sqcup m.v$
74:     Add $m$ to $R$
75: end upon
```

4.4 Correctness

We show that our primitives provide the same safety properties as writing and reading to simple quorums in the Crash-Stop model. First, we formally define quorum knowledge in the DCR context.

▶ **Definition 4** (Stable Properties)**.** A predicate on the history of an incarnation of a process (i.e., the sequence of events it has processed) is a *stable property* if it is monotonic (i.e., X being true of history h implies that X is true of any history with h as a prefix).

▶ **Definition 5.** If stable property X is true of some incarnation of a process, p, we say that incarnation of p *knows* X.

▶ **Definition 6** (Quorum Knowledge). We say that a quorum Q *knows* stable property X if, for all processes $p \in Q$, one of the following holds:

(1) p is DOWN,

(2) p is OPERATIONAL and knows X, or

(3) p is RECOVERING and either already knows X or will know X if and when it finishes recovery.

In our analysis of Algorithm 1, we are concerned with knowledge of two types of stable properties: knowledge of values and knowledge of incarnation IDs. An incarnation of a process knows value *val* if it has either executed UPDATE-STATE(*val*) or executed REBUILD-STATE with an ACQUIRE-REP message in the reply set sent by a process which knew *val*. Knowledge of a process's incarnation ID, i, is the stable property of having an entry in a crash vector for that process greater than or equal to i.

Next, we define crash-consistency on ACQUIRE-REP messages with crash vectors.

▶ **Definition 7** (Crash Consistency). A set of ACQUIRE-REP messages R is *crash-consistent* if $\forall s_1, s_2 \in R.\ s_1.v[s_2.f] \leq s_2.v[s_2.f]$.

Note that Definition 7, phrased in terms of crash vectors, is equivalent to the sending events of the ACQUIRE-REP messages being crash-consistent according to Definition 3.

▶ **Definition 8** (Quorum Promise). We say that a crash-consistent set of ACQUIRE-REP messages constitutes a *quorum promise* for stable property X if the set of senders of those messages is a quorum, and each sender knew X when it sent the message.

▶ **Definition 9.** If process p sent one of the ACQUIRE-REP message belonging to a quorum promise received by some process, we say that p *participated* in that quorum promise.

The post-condition of the loop on line 53 guarantees the crash-consistency of the reply set by discarding any inconsistent messages; the next loop guarantees that there is at most one message from each process in the reply set. Therefore, the termination of ACQUIRE-QUORUM (line 10) implies 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 *message*. This implies that whenever a process finishes recovery, it must have received a quorum promise showing that the participants in its recovery had that process's latest incarnation ID in their crash vectors.

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.

4.4.1 Safety

Finally, we are ready to state the main safety properties of our generic read/write primitives.

▶ **Theorem 10** (Persistence of Quorum Knowledge). *If at time t, some quorum, Q, knows stable property X, then for all times $t' \geq t$, Q knows X.*

Proof. We prove by strong induction on t' that the following invariant, I, holds for all $t' \geq t$: 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 $t' - 1 \geq t$, we show that I holds at time t'. The only step any process $p \in Q$ could take to falsify I is finishing recovery. If recovery began at or before time t, then because Q knew X, p must know X now that it has finished recovering. Otherwise, if it 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. Therefore, p must know X upon finishing recovery since it updates its crash vector and rebuilds its state using m.

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

▶ **Theorem 11** (Acquisition of Quorum Knowledge). *If process p receives a quorum promise for 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 a process receives a quorum promise for stable property X from quorum Q, then Q knows X.
2. If process p ever participated in a quorum promise for X at or before time t, and p is OPERATIONAL, then p knows X.

I holds vacuously at $t = 0$. We show that if I holds at time $t - 1$, it holds at time t:

First, we consider part 1 of I. If p has received a quorum promise, R, from quorum Q for X, then because R is crash-consistent, we know that *at the time they participated in R* no process in Q had participated in the recovery 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, such a process would have known the recovered process's new incarnation ID when it participated in R, and R would not have been crash-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 in R,
(2) are DOWN,
(3) are RECOVERING, or
(4) are OPERATIONAL and know X.

In the base case, no process in Q has yet participated in R. 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. If q finished recovering, it must have received a quorum promise showing that the senders knew its new incarnation ID. By quorum intersection, at least one of these came from some process $r \in Q$. 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. Because knowledge of values and incarnation IDs is transferred through ACQUIRE-REP messages, q knows X, completing this secondary induction.

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

Now, we consider part 2 of I. Suppose, for the sake of contradiction, that p is OPERATIONAL at time t and doesn't know X, but participated in quorum promise R for X at or before time t. Let Q be the set of processes participating in R. Since p does not know X, 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 incarnation ID (or a greater one) be R'. Let the set of participants in R' be Q'. By quorum intersection, there exists some $r \in Q \cap Q'$.

It must be the case that r participated in R' before R; otherwise by induction, when r participated in R', it would have known X, and then transferred that knowledge to the current incarnation of p (at time t). r couldn't have participated in R before time t, because then by part 2 of I, it would have known p's latest incarnation ID when participating in R, violating the consistency of R. However, r cannot participate in R at or after time t, either. Because p has received a quorum promise for its new incarnation ID at or before time t, by part 1 of I, Q' knows p's new incarnation ID. By Theorem 10, Q' continues to know this at all later times. Because $r \in Q'$, it must know p's incarnation ID, and thus cannot participate in R without violating its crash-consistency. This contradicts the fact that r participates in R and completes the proof that part 2 holds at time t. ◀

Since ACQUIRE-QUORUM(val) obtains a quorum promise for val, Theorem 11 implies quorum knowledge of val, and Theorem 10 shows that that knowledge will persist for all future time, subsequent ACQUIRE-QUORUMs and READ-QUORUMs will get a response from a process which knows val.

4.4.2 Liveness

ACQUIRE-QUORUM and READ-QUORUM terminate if there is some quorum of processes that all remain OPERATIONAL for a sufficient period of time.[2] 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 crash-consistent. Note that the termination of ACQUIRE-QUORUM implies the termination of the recovery procedure, RECOVER. Therefore, the same liveness conditions are required for ACQUIRE-QUORUM and for recovery termination.

We define a sufficient liveness condition, LC, below. It is a slightly weaker version of the network stability condition N_2 from [17]: the period in which processes must remain OPERATIONAL is shorter.

▶ **Definition 12** (Liveness Condition (LC)). Consider a process p executing either the ACQUIRE-QUORUM or READ-QUORUM function, ϕ, and consider the following statements:

1. There exists a quorum of processes, Q, all of which consume their respective messages sent from ϕ.
2. Every process in Q either
   a. remains OPERATIONAL during the interval $[T_1, T_2]$, where T_1 is the point in time at which ϕ was invoked and T_2 the earliest point in time at which p completes the consumption of all the responses sent by the processes in Q or
   b. becomes DOWN and remains DOWN during the same interval after p consumed its response.

If these two statements are true for *every* invocation of a ACQUIRE-QUORUM or READ-QUORUM function, then we say that LC is satisfied.

[2] We assume that the application-provided READ-STATE, UPDATE-STATE, and REBUILD-STATE functions execute entirely locally and do not block.

Our protocol implementing the group communication primitives is live if LC is satisfied. For LC to be satisfied, it is necessary that at most a minority of processes are DOWN *at any given time*. Otherwise, no process can ever receive replies from a quorum again.

5 Recoverable Shared Objects in DCR

In this section we demonstrate the benefits of our quorum communication primitives for DCR: generality and efficiency. We present protocols for two different shared objects: a multi-writer, multi-reader (MWMR) atomic register and a single-writer, single-reader (SWSR) atomic set. In both protocols, READ and WRITE are intended to be invoked serially.

The first protocol implements a shared, fault-tolerant MWMR atomic register in DCR. It is more efficient and has better liveness conditions than prior work. The second protocol implements a weaker abstraction – a shared, fault-tolerant SWSR atomic set. We use this set as a basic storage primitive to provide processes with access to their own *virtual stable storage* (VSS), an emulation of a local disk. This enables easy migration of protocols to DCR.

5.1 Multi-writer, Multi-reader Atomic Register

We present a protocol for implementing a fault-tolerant, recoverable multi-writer, multi-reader (MWMR) atomic register in DCR, which guarantees the linearizability of READs and WRITEs [19]. Our protocol is similar to the ABD protocol [3] but augments it with a recovery procedure. Its pseudo-code is presented in Algorithm 2. Timestamps are used for version control, as in the original protocol. A timestamp is defined as a triple $(z, i, v[i])$, where $z \in N$, $i \in [1..n]$ is the ID of the writing process, and $v[i]$ is the incarnation ID of that process. Timestamps are ordered lexicographically. By replacing each quorum write phase in the original protocol with our ACQUIRE-QUORUM function and each quorum read phase with READ-QUORUM, we guarantee that every successful write phase is visible to subsequent read phases, despite concurrent crashes and recoveries, thus preserving safety in DCR. The REBUILD-STATE function reconstructs a value of the register at least as new as the one of the last successful write that finished before the process crashed.

Discussion. The most recent protocol for fault-tolerant, recoverable, MWMR atomic registers is *RADON* [17]. The always-safe version of *RADON*, $RADON_R^{(S)}$, introduces an additional communication phase after each quorum write to check whether any of the processes that acknowledged the write crashed in the meantime. This increases the latency of both the READ and WRITE procedures. Also, our liveness conditions are weaker: our protocol is live if any majority of processes do not crash for a sufficient period of time, while $RADON_R^{(S)}$ requires a supermajority (3/4) of processes to not crash.

5.2 Virtual Stable Storage

Algorithm 3 presents a protocol for a fault-tolerant, recoverable, SWSR set, where the reader is the same as the writer. It guarantees that the values written by completed WRITEs and those returned in READs are returned in subsequent READs. Given the group communication primitives, its implementation is straightforward; the only additional detail is that values read during recovery should be written back to ensure atomicity (line 17).

Discussion. We can use this set to provide a *virtual stable storage* abstraction. It is well known that any correct protocol in CS can be transformed into a correct protocol in the

Algorithm 2 Multi-writer, multi-reader atomic register in Diskless Crash-Recovery.

```
    Volatile Local State:
        $(t, d) \leftarrow (t_0, d_0)$ ▷ Value of register

 1: procedure WRITE($d_{\text{new}}$)
 2: guard: $op$
    ▷ Get latest timestamp
 3:     $\Sigma \leftarrow$ READ-QUORUM
 4:     $(t_{\max}, d_{\max}) \leftarrow \max(\Sigma)$
    ▷ Write value
 5:     $t_{\text{new}} \leftarrow (t_{\max}.z + 1, i, v[i])$
 6:     ACQUIRE-QUORUM($(t_{\text{new}}, d_{\text{new}})$)
 7: end procedure

 8: procedure READ
 9: guard: $op$
    ▷ Get latest register value
10:     $\Sigma \leftarrow$ READ-QUORUM
11:     $(t_{\max}, d_{\max}) \leftarrow \max(\Sigma)$
    ▷ Write latest register value
12:     ACQUIRE-QUORUM($(t_{\max}, d_{\max})$)
13:     return $d_{\max}$
14: end procedure

15: function UPDATE-STATE($val$)
16:     if $val.t > t$ then
17:         $(t, d) \leftarrow val$
18:     end if
19: end function

20: function READ-STATE
21:     return $(t, d)$
22: end function

23: function REBUILD-STATE($\Sigma$)
24:     $(t, d) \leftarrow \max(\Sigma)$
25: end function
```

Algorithm 3 Single writer, single reader atomic set in Diskless Crash-Recovery.

```
    Permanent Local State:
        $owner$ ▷ Owner of set flag

    Volatile Local State:
        $S \leftarrow \{\}$        ▷ Local set

 1: procedure WRITE($s$)
 2: guard: $op \wedge owner$
 3:     ACQUIRE-QUORUM($\{s\}$)
 4:     $S \leftarrow S \cup \{s\}$
 5: end procedure

 6: procedure READ
 7: guard: $op \wedge owner$
 8:     return $S$
 9: end procedure

10: function UPDATE-STATE($val$)
11:     $S \leftarrow S \cup val$
12: end function

13: function READ-STATE
14:     return $S$
15: end function

16: function REBUILD-STATE($\Sigma$)
17:     ACQUIRE-QUORUM($\Sigma$)
18:     $S \leftarrow \bigcup \Sigma$
19: end function
```

CRSS model by having processes write every message they receive (or the analogous state update) to their local disk before sending a reply. By equipping each process with VSS, any correct protocol in the CRSS model can then be converted into a safe protocol in the DCR model, wherein processes write to crash-consistent quorums instead of stable storage.

6 Recoverable Replicated State Machines in DCR

We further extend our study of DCR to another specific problem: state machine replication (SMR). SMR is a classic approach for building fault-tolerant services [18,32] that calls for the service to be modeled as a deterministic state machine, replicated over a group of replicas. System correctness requires each replica to execute the same set of operations in the same order, even as replicas and network links fail. This is typically achieved using a consensus-based replication protocol such as Multi-Paxos [20] or Viewstamped Replication [22,29] to establish a global order of client requests.

We examined three diskless recovery protocols for SMR: Viewstamped Replication [22], Paxos Made Live [7], and JPaxos [16]. We found that each of these protocols suffers from the problem illustrated in the example at the beginning of Section 4: they use regular quorums of responses (instead of crash-consistent ones) when persisting critical data, which could violate their invariants. This can lead to operations being lost, or different operations being executed at different replicas, both serious correctness violations. We provide here brief explanations of the problems in each of these protocols. For more details on how the protocols work and complete traces, see our technical report [27].

Viewstamped Replication [29] is the first consensus-based SMR protocol. The original version of the protocol requires a single write to disk, during a view change. A recent VR variant [22] replaces the write to disk with a write to a quorum of replicas, in an attempt to eliminate the necessity for disks. However, it uses simple quorum responses, allowing the recovering replica to violate an important invariant: once a replica committed to take part in a new view, it will never operate in a lower view. As a result, an operation can complete successfully and then be lost after a view change.

Paxos Made Live [7] is Google's Multi-Paxos implementation. 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 replica must then wait to observe a full instance of successful consensus before participating. This 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., leader 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 [16], 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, certain failures during node recovery can cause the system to lose state and violate safety properties.

All of these protocols can be correctly migrated to DCR, with little effort, using VSS write operations, as explained in Section 5.2. This approach is straightforward, efficient, and requires no invasive protocol modifications.

7 Conclusion

This paper examined the Diskless Crash-Recovery model, where process can crash and recover but lose their state. We show how to provide persistence guarantees in this model using new quorum primitives that write to and read from *crash-consistent quorums*. These general primitives allow us to construct shared objects in the DCR model. In particular, we show a MWMR atomic register protocol requiring fewer communication rounds and weaker liveness assumptions than the best prior work. We also build a SWSR atomic set that can be used to provide each process with *virtual stable storage*, which can be used to easily migrate any protocol from traditional Crash-Recovery models to DCR.

Acknowledgments. We thank Marcos K. Aguilera for his comments on early drafts of this work, as well as Irene Zhang, the anonymous reviewers, and Jennifer L. Welch for their helpful feedback.

References

1 Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, April 2011.

2 Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. In *Proc. of DISC*, 1998.

3 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. of the ACM*, 42(1):124–142, January 1995.

4 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 *Proc. of DISC*, 2015.

5 Roberto Baldoni, Silvia Bonomi, Anne-Marie Kermarrec, and Michel Raynal. Implementing a register in a dynamic distributed system. In *Proc. of ICDCS*, 2009.

6 Roberto Baldoni, Silvia Bonomi, and Michel Raynal. Implementing a regular register in an eventually synchronous distributed system prone to continuous churn. *IEEE Trans. Parallel Distrib. Syst.*, 23(1):102–109, January 2012.

7 Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proc. of PODC*, 2007.

8 Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Failure detectors in omission failure environments. In *Proc. of PODC*, 1997.

9 Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proc. of ACSC*, 1988.

10 Eli Gafni and Dahlia Malkhi. Elastic configuration maintenance via a parsimonious speculating snapshot solution. In *Proc. of DISC*, 2015.

11 Rachid Guerraoui, Ron R. Levy, Bastian Pochon, and Jim Pugh. The collective memory of amnesic processes. *ACM Trans. Algorithms*, 4(1):12:1–12:31, March 2008.

12 Michel Hurfin, Achour Mostéfaoui, and Michel Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Proc. of SRDS*, 1998.

13 Leander Jehl, Tormod Erevik Lea, and Hein Meling. Replacement: Decentralized failure handling for replicated state machines. In *Proc. of SRDS*, 2015.

14 Leander Jehl, Roman Vitenberg, and Hein Meling. SmartMerge: A new approach to reconfiguration for atomic storage. In *Proc. of DISC*, 2015.

15 Andreas Klappenecker, Hyunyoung Lee, and Jennifer L. Welch. Dynamic regular registers in systems with churn. *Theor. Comput. Sci.*, 512:84–97, November 2013.

16 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, 2011.

17 Kishori M. Konwar, N. Prakash, Nancy A. Lynch, and Muriel Médard. RADON: Repairable atomic data object in networks. In *Proc. of OPODIS*, 2016.

18 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 1978.

19 Leslie Lamport. On interprocess communication. *Distributed Computing. Parts I and II*, 1986.

20 Leslie Lamport. Paxos made simple. *ACM SIGACT News 32*, 2001.

21 Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, March 2010.

22 Barbara Liskov and James Cowling. Viewstamped Replication revisited. Technical report, MIT, July 2012.

23 Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The SMART way to migrate replicated stateful services. In *Proc. of EuroSys*, 2006.

24 Nancy A. Lynch and Alexander A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proc. of DISC*, 2002.

25 Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

26 Michael Merritt and Gadi Taubenfeld. Computing with infinitely many processes under assumptions on concurrency and participation. In *Proc. of DISC*, 2000.

27 Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. Recovering shared objects without stable storage [extended version]. Technical Report UW-CSE-17-08-01, University of Washington CSE, August 2017.

28 Peter Musial, Nicolas Nicolaou, and Alexander A. Shvartsman. Implementing distributed shared memory for dynamic networks. *Comm. of the ACM*, 57(6), June 2014.

29 Brian M. Oki and Barbara H. Liskov. Viewstamped Replication: A new primary copy method to support highly-available distributed systems. In *Proc. of PODC*, 1988.

30 R.C. Oliveira, R. Guerraoui, and A. Schiper. Consensus in the crash recover model. Technical Report TR-97/239, EPFL, Lausanne, Switzerland, 1997.

31 D.S. Parker, G.J. Popek, G. Rudisin, A. Stoughton, B.J. Walker, E. Walton, J.M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. In *IEEE Trans. on Software Engineering*, 1983.

32 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 1990.

33 Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio P. Junqueira. Dynamic reconfiguration of primary/backup clusters. In *Proc. of USENIX ATC*, 2012.