Reliable, Efficient Recovery for Complex Services with Replicated Subsystems

Edward Tremel, Sagar Jha, Weijia Song, David Chu, and Ken Birman
Cornell University, Ithaca, NY, USA

***Abstract*—Applications with internal substructure are common in the cloud, where many systems are organized as independently logged and replicated subsystems that interact via flows of objects or some form of RPC. Restarting such an application is difficult: a restart algorithm needs to efficiently *provision* the subsystems by mapping them to nodes with needed data and compute resources, while simultaneously guaranteeing that replicas are in distinct failure domains. Additional failures can occur during recovery, hence the restart process must itself be a restartable procedure. In this paper we present an algorithm for efficiently restarting a service composed of sharded subsystems, each using a replicated state machine model, into a state that (1) has the same fault-tolerance guarantees as the running system, (2) satisfies resource constraints and has all needed data to restart into a consistent state, (3) makes safe decisions about which updates to preserve from the logged state, (4) ensures that the restarted state will be mutually consistent across all subsystems and shards, and (5) ensures that no committed updates will be lost. If restart is not currently possible, the algorithm will await additional resources, then retry.**

I. Introduction

We are seeing a shift from a *query-dominated cloud* in which most operations are read-only and use data acquired out-of-band, to a *real-time control cloud*, hosting increasingly complex online applications, in which near-continuous availability is important. Such needs arise in stream processing for banking and finance, IoT systems that monitor sensors and control robots or other devices, smart homes, smart power grids, smart highways, and cities that dynamically manage traffic flows, etc. These applications often have multiple subsystems that interact, and that bring safety requirements which include the need for fault-tolerance and consistency in the underlying data-management infrastructure.

Traditional transactional database methods scale poorly if applied naively [1]. Our work adopts a *state machine replication* model, using *key-value sharding* for scaling. Such models are relatively easy to program against and hence increasingly popular, but pose challenges when crashes occur.

To maintain the basic obligations of the state machine replication methodology, updates must be applied to replicas exactly once, in the same order, and should be durable despite damage a failure may have done. For a given replication factor the system should also guarantee recoverability if fewer than that number of crashes occur. Subsystems may being further constraints: numbers of cores, amounts of memory, etc. A further consideration is that datacenter hardware can exhibit correlated failures due to shared resource dependencies. To ensure high availability, replicas must be placed into distinct *failure correlation sets*.

Performance considerations further shape the design of modern cloud systems, which often migrate artificially intelligent behavior into the edge [2]. This may entail use of machine learned models for decision-making or event classification, as well as real-time learning in which new models are trained from the incoming data stream. For example, a smart highway might need to learn the behavior of vehicles, and adapt the acquired models as vehicles change their behavior over time. The large data sizes (photos, videos, radar, lidar) and intense time pressure (guidance is of little value if based on stale data) compel the use of accelerators, such as RDMA (which offloads data transport to hardware and achieves zero-copy transfers), NVM (which offers durable memory-mapped storage), GPU and TPU, and FPGA, without which applications would often be unable to meet performance demands [3, 4, 5].

The Derecho library [3] was created to support this new class of demanding edge applications. Derecho models the application as a collection of subgroups where each subgroup is partitioned into shards (subgroups can overlap, but shards of the same subgroup are disjoint). Each shard is a replicated state machine. The membership of the entire system is managed in a top-level group, which consists of all the nodes in the system. Figure 1 shows an example application. Derecho makes several key design decisions that are necessary to achieve high performance:

- Consensus off the critical path: Derecho adopts a virtual synchrony approach [6]. The top-level group membership moves through epochs (or views) where each epoch is a failure-free run of the system. Each failure triggers a reconfiguration (or view change) of the group membership. The view change involves agreement on pending updates and recomputation of the membership of each shard.
- Update all, read any single replica: An update is only committed in a shard if it has been logged at every non-failed member. Every replica has full state, enabling fast single-replica queries that do not interfere with updates. In this model, a shard can survive the failure of all but one member without losing any committed updates. This is in contrast to quorum-based protocols [7], where it suffices to update a majority of replicas, but where a query or a restart involves merging state from multiple replicas. Moreover, Derecho pipelines updates, such that each log consists of a prefix of committed updates followed by a suffix of pending updates. A reconfiguration results in

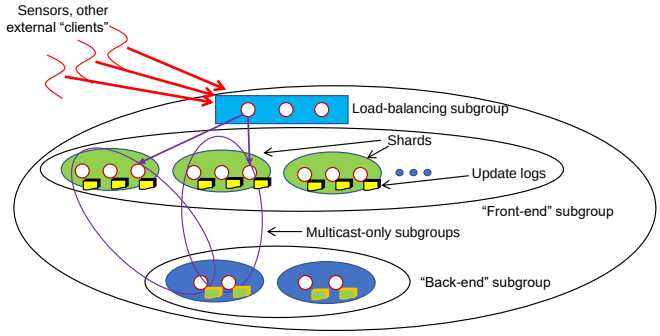


Fig. 1: A Derecho service spanning 16 (or more) machines and containing several subsystems that employ a mix of point-to-point RPC and multicast. The ovals represent subgroups and shards within which data is replicated. Independent use of state machine replication isn't sufficient: after a shutdown, components must restart in a *mutually-consistent* state.

a distributed log cleanup where updates that cannot be committed are discarded.

- Distributed logs: For safety, each shard member needs to log updates before they are committed. In this class of services, the "state of the application" is decentralized.

Services sometimes shut down and must later be restarted, for example when the application is migrated to different nodes, software is updated or the datacenter as a whole is serviced. Clearly we must recover each individual SMR subgroup or shard, but notice that the recovered states also need to correspond to a state the service *as a whole* could have experienced, while also preserving every committed update. This obligation is not unique to Derecho: systems like vCorfu [8] (the multi-log version of Corfu [9]) and Ceph [10] also have multiple subsystems that use sharding. Nonetheless, the problem has not previously been studied. For example, although the Derecho paper is detailed, it focuses on the efficiency of its protocols, their mapping to RDMA, and the resulting performance.

There are several factors that make restarting non-trivial:

- Failures during restart complicate the problem. We need to ensure safety under all circumstances and restore the system to a consistent, running state, equivalent to the last committed state before total failure.
- Some nodes that were once part of the system may never recover. Moreover, some restarting nodes may have failed in a view preceding the last view before the restart, in which case they will not be aware of the last membership of the top-level group. We need to determine the conditions under which a restart is possible and reconcile incomplete logs stored by shard members.
- We need to satisfy application constraints related to deployment. For example, shards may require that the members belong to different failure regions of the datacenter, impose a minimum on the number of members, and specify hardware configurations (such as number of cores, amount of memory, GPUs, etc).

The restart process should also be highly efficient to minimize application downtime. Thus we need to minimize the data transferred during restart and optimize data movement.

In this paper, we describe our restart algorithm for such systems, with configurable parameters as follows. Our algorithm requires the restarting service to designate a restart leader; it can be any restarting node. We model the failure characteristics of the nodes by organizing them into *failure correlation sets*. The application specifies the minimum number of failure correlation sets that the members of a shard should come from, for each shard of every subgroup. The application provides mappings from nodes to failure correlation sets through configuration files, making the process highly flexible; it can choose to distinguish nodes that belong to different racks or different regions of the datacenter altogether.

Our paper makes the following contributions:

1) Characterization of the state recovery problem for services composed of stateful subsystems, including a definition of correct recovery for replicated state machines that share a configuration manager.
2) An algorithm for restarting such a system from durable logs, including reasoning that argues why the algorithm is safe in the presence of any number of crashes, and live as long as any quorum of the last live configuration eventually restarts.
3) An algorithm that provably assigns nodes to shards in a way that satisfies deployment constraints and minimizes state transfer.
4) An experimental evaluation showing that a structured service can be recovered quickly and efficiently using this algorithm.

An implementation is available in the Derecho system.

In section II, we describe the restart problem at length, discussing our desirable goals for any algorithm that solves it. In section III, we discuss our restart algorithm and the accompanying algorithm for assigning nodes to shards while satisfying deployment constraints. In section IV, we reason about the correctness of the restart algorithm and prove the node assignment algorithm correct. We show the feasibility of our approach in section V and discuss related work in section VI. Finally, we summarize our findings and conclude in section VII.

II. Problem Description

The essence of our problem is that independent recovery of state-machine replicated components is not sufficient. SMR guarantees that a service with $2f + 1$ members can tolerate f crash failures. However a complex service with multiple subsystems and shards has many notions of f. For the service as a whole, Derecho's virtually synchronous membership protocol requires that half the members remain active from one view to the next; this prevents "split brain" behavior. But notice that in Figure 1 some shards have as few as 2 members. A log instance could be lost in a crash, hence such a shard must not accept updates if even a single member has crashed.

We can distinguish two cases. One involves continued activity of a service that experiences some failures, but in which many nodes remain operational. This form of *partial* failure has been treated by prior work, including the Derecho paper. In summary, if the partial failure creates a state in which safety cannot be maintained, the service must halt ("wedge") and cannot install a new view or accept further updates until the damage has been repaired.

The second case is our focus here: a full shutdown, which may not have been graceful (the service may not have been warned). To restart, we must first determine the final membership of the entire service, and the mapping of those nodes to their shard memberships in the restarted service. Then we must determine whether all the needed durable state is available, since recovery cannot continue if portions of the state are lacking, even for a single shard. Furthermore, intelligent choices must be made about the mapping of nodes to shard roles in the restarted service. On the one hand, this must respect constraints. Yet to maximize efficiency it is also desireable to minimize "role changes" that entail copying potentially large amounts of data from node to node.

In what follows, we will describe the restart problem and our algorithm for its solution in terms of a more generic system, with the hope that our techniques will be useful even in systems where Derecho is not employed.

A. System Setup

We consider a distributed system of nodes (i.e. processes) organized into subgroups partitioned into shards, in which each shard implements a virtually synchronous replicated state machine. In general, we will refer to a shard without mentioning which subgroup it belongs to, unless the distinction is important for clarity. Each shard maintains a durable log of totally ordered updates to its partition of the system state, and an update is considered committed once it is logged at every replica in the current view. As is standard in the virtual synchrony approach [6], each update records the view in which it was delivered. Also, each reconfiguration (view change) event requires every node to commit to an *epoch termination* decision which must contain, at a minimum, the highest update sequence number that can be committed in each shard, as well as the ID of the view that it terminates.

We believe this model to be quite general. Obviously, it is a natural fit for services implemented using Derecho, but it can also be applied to the materialized stream abstraction in vCorfu [8]. A vCorfu stream abstracts the action of applying a sequence of updates to a single replicated object (what we would call a shard). Moreover, vCorfu has multiple subsystems: it stores the system's configuration in a separate layout server, rather than having replicas store their own configuration. Turning to the Ceph file system [10], we find a metadata service, a cluster mapping service, and a sharded SMR-replicated object store (RADOS). Again, the requirements are analogous to the ones we described for Derecho, with the cluster map playing the role of the view. To our knowledge, neither vCorfu nor Ceph currently addresses the issue of consistency across different shards and subsystems in the event of a full shutdown; our methods would thus strengthen the recoverability guarantees offered by these systems.

B. The Restart Problem

Our task is to ensure that the committed state of this system can be recovered in the event that every node in the system crashes in a transient way. This could be the result of a power failure or network disconnection, or an externally-mandated shutdown caused by datacenter management policies. When the system begins restarting after such a failure, we can assume that most of the nodes that crashed will resume functioning and can participate in the restart process. However, some nodes may remain failed. The system should be able to restart as long as enough of its former members participate in the restart process to guarantee that its state has been correctly restored.

Specifically, we need to restore the system to a consistent, running state, that is equivalent to its last committed state before the total failure. The restarted system must also have the same fault tolerance guarantees as it did before. This means that the restarted distributed service must (1) include every update in every shard that had reached a durably-committed state before the crash, (2) adopt a configuration that is the result of a valid view-change procedure from the system's last installed configuration, and (3) assign nodes to shards such that each shard meets its constraint of having nodes from different failure correlation sets.

The service must also be resilient against failures during the restart process, since the same transient crashes that caused it to stop can also occur during restart. It must tolerate the failure of any node in the system, detect it, and revert the system to a safe state until recovery can continue. Recovery must be able to continue from any intermediary state.

We assume that some simple external process triggers the restart procedure, such as a datacenter-management system that re-runs each interrupted program after a shutdown event. As a preliminary design choice, we will also assume that the restart procedure will be leader-based. The restarting system's first task, then, is to choose a *restart leader*. While we could elect a restart leader using standard techniques, we found it simpler and just as effective to use a preconfigured list of restart leaders installed on all nodes in the system (e.g. through a settings file). We have designed our protocol such that any node that was a member of the system at any time can serve as the restart leader, so the choice of restart leader is arbitrary and does not depend on the state of the system at the time of the total failure. As we will see in section IV-B, this also means that it is easy for another node to take over for the restart leader if it fails during recovery.

In order to restart to a consistent state, several subproblems must be addressed. First, when the restart leader starts up, it does not know whether it was a member of the last installed configuration, or whether it crashed much earlier but was nonetheless set as the restart leader; thus, its logs of both system state and the group membership could be arbitrarily out of date. Second, when the restart leader communicates with

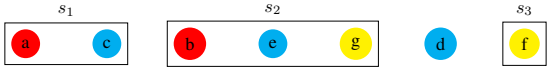
other restarting nodes, it must determine whether those nodes' configuration and state data is newer or older than its own, and whether it represents the last known state of the system, without knowing in what order the other nodes crashed. Third, for each node that restarts and has logged state updates, the restart leader must determine which updates in that log might have been externally visible and acted upon, and which were still in-progress and might never have reached a majority of replicas. Answering this question requires knowing what configuration was active at the time the update was logged, and what configuration was active at the time the system crashed. Finally, during the restart process any node could experience another transient crash, including the restart leader itself, and these crashes should not result in the system restarting in an inconsistent state or prevent the system from restarting when it has a sufficient number of healthy replicas.

The restarted system must also install a configuration that meets each shard's fault-tolerance constraints. To avoid shard shutdowns due to correlated failures, each shard is statically configured to require a minimum number of nodes from different failure correlation sets. Here, a distinction between shards of different subgroups is important, since only shards of the same subgroup are disjoint. Given a number of restarted nodes and their failure correlation sets, the restart leader must not only partition them between each subgroup's shards, but it must also (1) satisfy the minimum number of nodes required from different failure correlation sets for each shard, (2) assign as many nodes as possible to their original shards, in order to minimize the number of state transfers between nodes, and (3) compute the new assignment in a timely manner. Section II-C gives a detailed example of what is required.

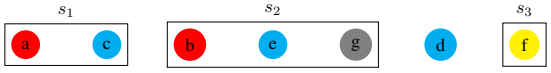
The log-recovery system we describe here addresses all of these concerns, and restarts the system as efficiently as possible by allowing each shard to complete state transfer operations in parallel.

C. Failure-Domain-Aware Assignment

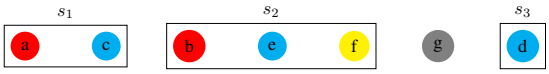
Suppose that a system has failure correlation sets f_1, f_2, f_3, such that f_1 contains nodes a and b, f_2 contains nodes c, d, and e, and f_3 contains nodes f and g. It has just one subgroup with three shards s_1, s_2, s_3, which require 2, 3, and 1 node(s) from different failure correlation sets respectively. A valid initial configuration for this system would be $s_1 = \{a, c\}$, $s_2 = \{b, e, g\}$, $s_3 = \{f\}$, leaving d unassigned to any shard. This can be represented in the following diagram, in which colors correspond to failure correlation sets:



Now suppose a shutdown occurs and all nodes except g restart. Shard s_2 is no longer in a valid configuration because it has 1 less node than it requires, but it would not suffice for the restart leader to simply add the unassigned node d to the shard because d is from the same failure correlation set as c.



An optimal reassignment is to move f from s_3 to s_2, and add d to s_3, resulting in the post-restart configuration $s_1 = \{a, c\}$, $s_2 = \{b, e, f\}$, $s_3 = \{d\}$. This reassigns only 2 nodes to new roles, which is the minimum that can be achieved while satisfying each shard's requirements.



III. Restart Algorithm

Having established the parameters of the restart problem, we now present our algorithm for solving it. At a high level, this algorithm has seven steps:

1) Find the last-known view by inspecting persistent logs
2) Wait for a quorum of this view to restart
3) Find the longest replicated state log for each shard
4) Compute new shard assignments and complete epoch termination from the last view, if necessary
5) Trim shard logs with conflicting updates
6) Update replicas with shorter logs
7) Install the post-restart view

However, this is not a linear process, because failures at any step after 2 can force the algorithm to return to step 2 if the quorum is lost. Also, in practice, steps 1-3 are executed concurrently by the restart leader, because it can gather information about the longest update log available for each shard while it is waiting to reach a restart quorum.

In order for log recovery to be possible, we must add a few requirements to the system described in Section II-A. First, during a reconfiguration, all nodes which commit to a new view must log it to nonvolatile storage before installing it. Furthermore, in order to ensure that no updates are used in the restarted state of the system that would have been aborted by the epoch termination process, live nodes must log each epoch termination decision to persistent storage before acting upon it. Before committing to a new view, the new members of each shard must download and save the epoch termination information for the prior view in addition to the logged updates that they download during the state-transfer process.

The pseudocode for our algorithm is shown in Algorithms 1, 2, 3, and 4, where Algorithms 1 and 2 show the code that runs on the restart leader, Algorithm 3 shows the code that runs on a non-leader node, and Algorithm 4 shows the STATE_TRANSFER function that is common to both nodes. For brevity, we have factored out the leader's failure-handling code into a macro called HANDLE_FAILURE, which should be inserted verbatim wherever it is named.

In our pseudocode's syntax, the dot-operator accesses members of a data structure by name, and the bracket operator accesses members of a map by key, as in C++ or Java. Note that there are three kinds of integer identifiers: node IDs or NIDs, shard IDs or SIDs, and view IDs or VIDs. Each node has a globally unique node ID, and, as is common in virtual synchrony, view IDs are unique and monotonically increasing. Shard IDs are unique identifiers assigned to each

Algorithm 1 The restart leader's behavior, part 1

```
 1: V_c ← READ(view_log)
 2: restarted ← {nid_me}
 3: u_e ← READ(update_log).end
 4: LL ← {V_c.my_sid → (nid_me, u_e.seqno)}
 5: ET ← READ(epoch_termination_log)
 6: while ¬ QUORUM(V_c, restarted) do
 7:     (V_i, nid_n, sid, seqno) ← RECEIVE from n
 8:     restarted ← restarted ∪ {nid_n}
 9:     if V_i.vid > V_c.vid then
10:         V_c ← V_i
11:         WRITE(view_log, V_c)
12:         ET ← {}
13:     if LL[sid].seqno < seqno then
14:         LL[sid] ← (nid_n, seqno)
15:     et ← RECEIVE from n
16:     if et ≠ {} ∧ et.vid = V_c.vid then
17:         ET ← et
18:         WRITE(epoch_termination_log, ET)
19: V_r ← CHANGE_VIEW(V_c, restarted)
20: if ET = {} then
21:     ET.vid ← V_c.vid
22:     for all s ∈ V_r.subgroups do
23:         ET.last[s.sid] ← LL[s.sid].seqno
24: sent ← {}
25: for all s ∈ V_r.subgroups do
26:     for all nid_n ∈ s.members do
27:         success ← SEND(V_r, ET, LL[s.sid].nid) to n
28:         if ¬success then
29:             HANDLE_FAILURE(nid_n, sent)
30:         sent ← sent ∪ {nid_n}
```

Algorithm 2 The restart leader's behavior, part 2

```
31: if ET.vid = u_e.vid then
32:     success ← SEND(∅) to LL[V_r.my_sid].nid
33:     trim_seqno ← ET.last[V_r.my_sid]
34: else
35:     success ← SEND(u_e.vid) to LL[V_r.my_sid].nid
36:     trim_seqno ← RECEIVE from LL[V_r.my_sid].nid
37: if ¬success then
38:     HANDLE_FAILURE(LL[V_r.my_sid].nid, restarted)
39: TRUNCATE(update_log, trim_seqno)
40: success ← STATE_TRANSFER(
        LL[V_r.my_sid].nid, nid_me, V_r)
41: if ¬success then
42:     HANDLE_FAILURE(LL[V_r.my_sid].nid, restarted)
43: sent ← {}
44: for all nid_n ∈ V_r.members do
45:     success ← SEND("Prepare") to n
46:     if ¬success then
47:         HANDLE_FAILURE(nid_n, sent)
48: for all nid_n ∈ V_r.members do
49:     SEND("Commit") to n
50: WRITE(view_log, V_c)
51:
52: procedure HANDLE_FAILURE(nid, notify_set)
53:     restarted ← restarted − {nid}
54:     for all nid_m ∈ notify_set do
55:         SEND("Abort") to m
56:     if ¬ QUORUM(V_c, restarted) then
57:         goto 6
58:     else
59:         goto 19
```

shard (globally, across all subgroups) of the system. In the following sections, we will explain the details of the algorithm, which should make the pseudocode more clear.

A. Awaiting Quorum

The restart leader's first operation is to read its logged view, which becomes the first "current" view, V_c, and its logged epoch termination information, which becomes the currently-proposed epoch termination, ET. It then begins waiting for other nodes to restart and contact it; non-leader nodes will contact the preconfigured restart leader as soon as they restart and discover that they have logged system state on disk.

When a non-leader node contacts the leader, it sends a copy of its logged view, V_i, its node ID, the ID of the shard it was a member of during V_i, and the sequence number of the latest update it has on disk. The joining node may optionally then send a logged epoch termination structure, if it has one that is as new as its logged view. The leader updates V_c and possibly ET if the client's view and epoch termination are newer, and uses data structure LL (a map from shard IDs to pairs of node IDs and update sequence numbers) to keep track of the location of the longest log for each shard. Note that sequence numbers from later views are always ordered after sequence numbers from earlier views.

After each node restarts, the leader checks to see if it has a *restart quorum*. A restart quorum consists of a majority of the members of the system in the last known view that includes at least one member of every shard from that view. In addition, the restart leader must be able to install a new post-restart view in which the entire group has at least $f+1$ replicas to meet the overall fault-tolerance threshold, and each shard is populated by nodes that meet its failure-correlation requirements. Note that the post-restart view can add new members that were not part of the last known view, since nodes that failed in an earlier view but restarted after the system-wide failure can still participate in the recovery process.

Once the leader has reached a restart quorum, if the newest epoch termination structure it has discovered is from an older view than V_c, it makes its own decision about how to terminate V_c's epoch. Specifically, it synthesizes an epoch termination structure by taking the sequence number of the latest update for each shard, and marking it with the same VID as V_c. It then computes V_r, the next view to install after restarting.

In practice, the leader waits for a short "grace period" after a quorum is achieved to allow nodes that restarted at a slightly slower rate to be included in V_r. This makes it less likely that V_r will require many node reassignments (and hence state transfers), and has only a minor effect on restart time.

B. Assigning Nodes to Shards

When testing for a restart quorum and computing V_r, the leader must determine an optimal assignment from nodes to shards. Since the shards of a subgroup must be disjoint, it can consider each subgroup individually. For each subgroup, the

Algorithm 3 A non-leader node's behavior

```
1:  V_c ← READ(view_log)
2:  et ← READ(epoch_termination_log)
3:  u_e ← READ(update_log).end
4:  SEND(V_c, nid_me, V_c.my_sid, u_e.seqno) to leader
5:  if et ≠ {} ∧ et.vid = V_c.vid then
6:      SEND(et) to leader
7:  commit ← ⊥
8:  while ¬commit do
9:      (V_r, ET, nid_ℓ) ← RECEIVE from leader
10:     et ← ET
11:     if et.vid = u_e.vid then
12:         success ← SEND(∅) to ℓ
13:         trim_seqno ← et.last[V_c.my_sid]
14:     else
15:         success ← SEND(u_e.vid) to ℓ
16:         trim_seqno ← RECEIVE from ℓ
17:     if ¬success then
18:         continue
19:     TRUNCATE(update_log, trim_seqno)
20:     success ← STATE_TRANSFER(nid_ℓ, nid_me, V_r)
21:     if ¬success then
22:         continue
23:     p ← RECEIVE from leader
24:     if p = "Prepare" then
25:         d ← RECEIVE from leader
26:         commit ← (d = "Commit")
27: V_c ← V_r
28: WRITE(view_log, V_c)
```

Algorithm 4 The state-transfer function

```
1:  function STATE_TRANSFER(nid_ℓ, nid_me, V_r)
2:      if nid_ℓ = nid_me then
3:          UL ← READ(update_log)
4:          for all n ∈ V_r.shards[V_r.my_sid] do
5:              vid_n ← RECEIVE from n
6:              if vid_n ≠ ∅ then
7:                  seqno_n ← FIND_MAX(UL, vid_n).seqno
8:                  succ_1 ← SEND(seqno_n) to n
9:              seqno_e ← RECEIVE from n
10:             succ_2 ← SEND({UL[seqno_e], ... UL.end}) to n
11:             if ¬succ_1 ∨ ¬succ_2 then
12:                 return ⊥
13:     else
14:         u_e ← READ(update_log).end
15:         success ← SEND(u_e.seqno) to ℓ
16:         if ¬success then
17:             return ⊥
18:         {u_{e+1}, u_{e+2}, ..., u_ℓ} ← RECEIVE from ℓ
19:         APPEND(update_log, {u_{e+1}, u_{e+2}, ..., u_ℓ})
20:     return ⊤
```

leader computes the assignment of nodes to shards in V_r by solving an instance of the min-cost flow problem [11].

It first creates a bipartite graph from shards to failure correlation sets as follows: For each shard there is a vertex s_i, and for each failure correlation set (FCS) there is a vertex fcs_j. There is one "shard" vertex u representing unassigned nodes, one source vertex, and one sink vertex. If m_i is the required number of nodes from different failure correlation sets for shard i, then there is an edge from the source vertex to s_i with cost 0 and capacity m_i. An edge with cost 0 and capacity 0 extends from the source vertex to u. An edge extends from each shard vertex s_i to each FCS vertex fcs_j, with cost 0 if shard i contained a node from FCS j in V_c, cost 1 otherwise, and capacity 1. For vertex u, these edges always have cost 0 and capacity 1. Finally, there is an edge from each FCS vertex fcs_j to the sink vertex with cost 0 and capacity equal to the number of nodes in FCS j in V_r.

The leader solves min-cost flow on the generated bipartite graph, increasing flow along augmenting paths until all shard vertices s_i have at least m_i flow and a solution is generated, or no augmenting path can be generated for the graph. If a solution is generated, then the leader translates that solution into a node assignment, where shard i is assigned one node from failure correlation set j if an edge contains flow between vertices s_i and fcs_j. If min-cost flow halts without a solution, then there is no solution that satisfies m_i for all shards, and there is not yet a restart quorum.

C. Completing Epoch Termination

For each shard, the restart leader sends to each node that will be a member in V_r the identity of the node on which the latest update for that shard resides (denoted node ℓ), as well as V_r itself and the epoch termination information.

When sending this information to node n, the restart leader might discover that n has crashed because it does not respond to the leader's connection attempts (we assume TCP-like semantics for our network operations). In this case, the leader removes n from the set of restarted nodes, sends an "Abort" message to all the nodes that have already received its message, and recomputes whether it has a restart quorum. If there is still a restart quorum, the leader recomputes V_r and starts over at sending ET and V_r to each live node. If not, it returns to step 2 and waits for additional nodes to restart.

Meanwhile, when a non-leader node receives V_r, ET, and nid_ℓ, it compares ET's view ID to the view ID associated with its last logged update. If these IDs match, the node completes epoch termination by deleting from its update log any updates with a sequence number higher than the last commit point for its shard. If the epoch termination structure is from a later view, though, all the updates in the node's log are from an earlier view that might have had its own epoch termination. In order to ensure that it also trims any updates that were aborted by the earlier epoch termination, the node contacts node ℓ and sends it the VID of its last logged update. Node ℓ, upon receiving this message, inspects its update log and finds the last update with that VID, then replies with that update's sequence number. The sending node then deletes from its log any updates with a higher sequence number. (Node ℓ's behavior in this exchange is implemented in the STATE_TRANSFER function).

D. Transferring State

Once each node, including the leader, has truncated from its log any updates that would have aborted, it must download

any committed updates that are not in its log. Each node that has been designated as the location of the longest log must, conversely, listen for connections from the other nodes that will be members of its shard in V_r and send them the updates they are missing. This is shown in the STATE_TRANSFER function in Algorithm 4. In this phase, a non-leader node may discover that the node with the longest log has failed when it attempts to contact it. In that case, the node can conclude that the V_r it has received from the leader will not commit, and return to waiting to receive a new V_r and longest-log location from the leader.

E. Committing to a Restart View

When a non-leader node finishes its state transfer operations, it awaits a "prepare" message from the leader. Meanwhile, when the leader has finished its own state transfer operations, it begins sending "prepare" messages to each node. If it discovers while sending these that a node has crashed (because the connection is broken), it sends an "abort" message to all nodes that it has already sent "prepare" messages to, and recomputes the post-restart view to exclude the crashed node. The leader might then discover that it no longer has a sufficient quorum for restart without the crashed node, in which case it returns to step 2 and waits for additional nodes to restart. If it still has a quorum, however, the leader can return to step 3, calculating the new shard membership and sending the new V_r and longest log location to all nodes. Once the leader has successfully sent "prepare" messages to all nodes in V_r, it can send a "commit" message to all of them confirming that this view can be installed. Once a non-leader node receives the leader's commit message, it can install V_r and begin accepting new messages and committing new updates. At this point, the restart leader no longer has a leader role, and all future failures and reconfigurations can be handled by the normal view-change protocol for a running system.

IV. Analysis

We will now prove that this protocol satisfies the goals we set out in section II-B. We first show that the protocol is correct in the case where there are no failures during the restart process, and then show that failures of any node do not affect its correctness.

Regardless of which view the restart leader has logged on disk when it first starts up, it is guaranteed to discover the last view that was installed in the pre-crash system before it exits the await-quorum loop, because a restart quorum requires a majority of nodes from the current view V_c to contact it. The view-change protocol in virtual synchrony requires a majority of the members of the current view to be members of the next view, which means that if the restart leader starts with some obsolete view V_k, a majority of members of V_k were also members of V_{k+1}, and the restart leader will discover at least one member of V_{k+1} by waiting for a majority of members of V_k. When a member of V_{k+1} restarts, it will send V_{k+1} to the leader, which will then use V_{k+1} as V_c and begin waiting for a majority of V_{k+1}'s members. If V_{k+1} is not the latest view, then by the same logic, the leader is guaranteed to discover V_{k+2} on at least one of the members of this majority. Thus, the leader must have discovered and installed the last known view V_ℓ by the time it has satisfied the quorum condition of contacting a majority of V_c.

Furthermore, by the time the leader exits the await-quorum loop, it is guaranteed to discover at least one log containing all committed updates for each shard in the system. This is because an additional condition of a restart quorum is that the leader must contact at least one member of each shard according to V_c. As we have just shown, V_c must equal V_ℓ before the majority condition of the quorum can be satisfied, so the leader will contact at least one member of each shard in V_ℓ. Since updates that commit in a view are by definition logged on every member of a shard in that view, any node that was a member of a shard in V_ℓ will have a log containing all committed updates for that shard up to the point of the total crash. Thus, every shard will have a designated longest-log location that contains all of its committed updates by the time the leader exits the await-quorum loop. Recovery into a mutually consistent state follows because membership epochs are totally ordered with respect to SMR events in shards or subgroups: the end of each epoch is a *consistent cut* [12].

The epoch termination decision ET that the leader sends out after achieving quorum is guaranteed to preserve any decision made by the group prior to the crash, and to include only updates that were safe to commit. Since the system's epoch termination process (as augmented in section III) requires all members of a view to log the epoch termination decision before acting on it, by the time the leader reaches a majority of V_ℓ, it must find at least one copy of the epoch termination information that was computed for V_ℓ if any node acted upon it. Using this epoch termination structure as ET preserves the decision made by V_ℓ's reconfiguration leader about which updates to include. Conversely, if the leader does not find any epoch termination information for V_ℓ, then no node had yet delivered or aborted any updates that were in-progress at the time of the crash. This means it is safe for the restart leader to construct ET using the longest sequence of updates that is available on at least one node in each shard, and unilaterally decide to commit any pending updates at the tail of that log. Before any node installs a view in which those updates are committed (V_r), the state transfer process ensures that any pending updates are fully replicated to all members of their shard. Thus, for each shard, every update up to the last commit point in ET will be present on all members of that shard in the new view, which is the same guarantee provided by the epoch termination process during a normal run of the system.

Finally, the post-restart view V_r that the leader installs is guaranteed to have the same stability and durability guarantees as any other view in the running system. As we just showed, all nodes that will be members of a shard in V_r will have the exact same update log for that shard before V_r is installed, which means that the updates committed in V_r are just as fault-tolerant as updates in any prior view. V_r itself is also durable, and guaranteed to be recovered by a future restart leader during

the recovery process, because a majority of members of V_ℓ are also members of V_r.

A. Tolerance of Failures of Non-Leaders

Our approach to failed non-leader nodes is to treat them as nodes that have not yet restarted. Upon detecting a failure at any point after reaching a restart quorum, the leader removes the failed node from its $restarted$ set, and recomputes both V_r and whether it has a restart quorum. By sending an "abort" message to all other nodes that may already have received V_r, the leader ensures that they will return to waiting for V_r and the epoch termination information. Regardless of how many times nodes fail and restart during the restart process, the leader still cannot proceed past the await-quorum loop until it has reached a restart quorum, which means it must reach at least one node from each shard that has all the committed updates for that shard. Since nodes never truncate updates from their logs that had actually committed in V_ℓ (due to the correctness of the epoch termination procedure), and committed updates were present on every member of their shard in V_ℓ, this will always be possible as long as enough members of V_ℓ eventually restart.

It is safe for the nodes that received ET and V_r from the leader before it detected a failure to begin the epoch termination and state transfer process, because at the point the leader started sending V_r it had reached a restart quorum. This means that ET only included updates that were safe to commit, and only excluded updates that had definitely aborted. Although V_r will change whenever there is a failure, the only way that ET could change after a failure is to include or exclude a different number of pending-but-uncommitted updates at the tail of a shard's log, and that will only happen if the node that failed was the location of the longest log for a shard. In that case, the new ET may include fewer of the uncommitted updates at the tail of the shard's log, but it is equally safe to abort these updates, since they had not yet committed at the end of V_ℓ. Nodes that had downloaded some of these updates at the time of the failure will simply truncate them when they re-run the epoch termination process.

The two-phase commit at the end of the state-transfer process ensures that all of the nodes in V_r are still live and have finished state transfer before any of them can commit to V_r. This ensures that no node can begin acting on V_r until all of the updates committed by ET are fully replicated.

B. Tolerance of Failure of the Leader

Much of our restart protocol seems to depend on correct operation of the restart leader, but in fact it can tolerate the failure of the restart leader: a subsequent restart leader would always select a state that is a safe extension of the state of the original leader (in fact it will be the identical state if the original leader's proposal might have been acted upon, and otherwise will be a safe choice with respect to the state the system was in when it crashed). One caveat is that our solution is correct only with a single leader running at a time. Since no fault-tolerant configuration management system is yet in place while the system is restarting, choosing a restart leader with an election protocol would be quite difficult. However, a small amount of manual configuration can to be used both to choose the initial leader and to select one to take over if the initial one fails. This can be accomplished by, for example, specifying both a default restart leader and an ordered list of fallback restart leaders in a configuration file. Handling the failure of the leader in an efficient manner may also require some manual intervention, specifically in the case where the leader fails during the await-quorum loop, because non-leader nodes can expect to wait a rather long time for the leader to reach a quorum (depending on how long it takes nodes in the system to restart after a total crash). They can eventually conclude that the leader has failed if it does not send ET after a suitably long timeout, but the restart process can complete faster if a system administrator or other outside process forcibly restarts them if the leader fails while awaiting a quorum. Failures of the leader during the 2-phase commit are easier to detect, because the leader should send the prepare and commit messages shortly after sending the ragged trim information, so the non-leader nodes can safely use much shorter timeouts on these messages.

When non-leader nodes detect that the leader has failed, they restart the recovery process using the new restart leader. This means that the new restart leader receives all of the same view, epoch-termination, and update-log information as the previous restart leader, and will reach the same conclusions. It will still wait for majority of members of each view it discovers to restart, meaning it must discover the last known view before it concludes that it has a restart quorum. If the previous restart leader was in fact required to achieve a quorum (because, for example, it was the only member of some shard in V_ℓ), then the new restart leader must wait for it to restart and rejoin the system as a non-leader.

C. Correctness of Node Assignment to Shards

Next, we prove that our min-cost flow algorithm finds a node assignment that satisfies each shard's required number of nodes from different failure correlation sets, given that a node assignment exists that obeys this constraint. We also show that our algorithm is optimal, generating an assignment where a minimal number of nodes are moved to shards they were not previously a part of. Thus, we minimize time spent on state transfer between old and new members of each shard.

We prove correctness by reduction to min-cost flow. Our solution is correct if it finds a feasible node assignment given that one exists. Given capacities of edges from the source vertex to shard vertices, shard vertices to failure correlation set vertices, and failure correlation set vertices to the sink vertex, any feasible flow in the graph can be translated into a feasible assignment of nodes to shards. Each shard receives exactly the number of nodes from different failure correlation sets it requires, because that is the capacity of the edge from the source to the shard vertex. No node from any failure correlation set is assigned to more than one shard, because the capacity of the edge from the failure correlation set vertex to the sink is number of nodes in that failure correlation set.

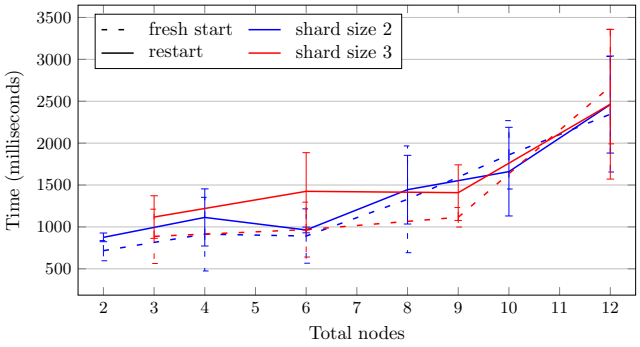


Fig. 2: Total time to start or restart a service. Error bars represent 1 standard deviation.

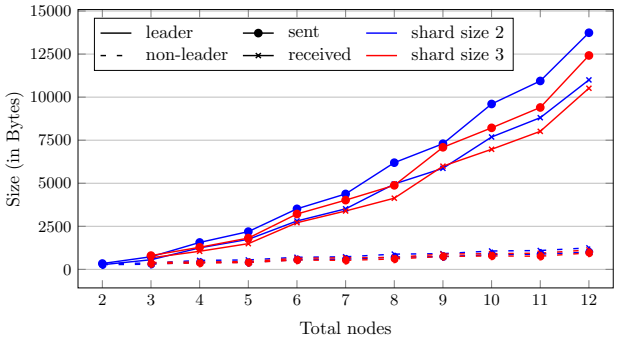


Fig. 3: Total metadata sent/received during the restart process.

All nodes assigned to any one shard are from different failure correlation sets, because the capacity of the edges from shard vertices to failure correlation set vertices is always 1. Thus a solution to min-cost flow is a solution to the node assignment problem. In fact, any solution to the node assignment problem can also be translated into a flow.

Furthermore, the solution to min-cost flow represents an optimal node assignment. We defined optimality above; a solution is optimal if it minimizes the number of nodes whose shard membership changes. By definition, the solution to min-cost flow is a flow that minimizes the cost along all its edges. Costs along edges are 0 except for edges from shard vertices to failure correlation set vertices, where no member from the failure correlation set belonged in the shard in the previous view. That is the definition of optimality. Thus any solution to min-cost flow is optimal.

Note that we opted to reduce to min-cost flow, which can be solved in polynomial time, instead of integer linear programming, which can be used to satisfy more generic constraints but might not find a solution efficiently.

D. Efficiency and Generality

Our restart protocol is designed for a particular form of state machine replication (the one implemented by Derecho), which allows us to take advantage of some efficiencies built into this SMR protocol. Specifically, Derecho's SMR enforces a read quorum of 1 within each shard, which means that reading the log of one up-to-date replica is sufficient to learn the entire committed state of that shard. Thus, the restart quorum only requires a single member of each shard from the last known view, and when new or out-of-date replicas are added to a shard during restart, they only need to contact and transfer state from a single up-to-date member. Furthermore, uncommitted updates only occur at the tail of a log, and there are no "holes" in the committed prefix of the log because updates are only aborted during a reconfiguration (which also trims them from the log). This allows us to easily make the correct decision about whether to accept these updates during recovery: they can safely be committed unless a logged epoch termination decision is found that proves they will be aborted.

Nevertheless, our protocol could be applied to other forms of SMR with a few relaxations of these optimizations. For example, a read quorum > 1 would merely increase the size of the restart quorum, as long as reconfiguration was still handled via virtual synchrony. In a system with a per-shard read quorum of r_i, the restart leader would need to contact at least r_i members of shard i in the current view in order to ensure it found both the next view (if one exists) and the longest sequence of committed updates in shard i; the restart quorum would include a read quorum of every shard in the last known view. Any nodes added to a shard in the restart view would also need to contact all the members of the most-recent read quorum in order to complete state transfer.

Some SMR systems, such as vCorfu [8], separate configuration information from the replicated state itself, using a separate "layout" service and "data" service. In this case, our protocol would need to explicitly separate step 1 (finding the last configuration) from step 3 (finding the longest log), rather than executing them concurrently. The restart leader would first need to contact a quorum of the layout service in order to find the last active configuration, then use that configuration to compute and wait for a restart quorum of the data service.

V. Experiments

We have implemented our restart algorithm as part of the Derecho library, and in this section we measure its performance when restarting sample Derecho applications. All experiments were carried out on our local cluster, which contains 12 servers running Ubuntu 16.04, using SSD disks for storage. In summary, we found that our recovery logic scales well, and adds only a small delay compared to the costs of process launches and initial Derecho platform setup.

Our first experiment was a straightforward end-to-end benchmark. We used our algorithm to restart a simple Derecho service with a single subgroup and shards of 2 or 3 nodes each, after an abrupt crash in which all nodes failed near the same time, and measured the time from when the restart leader launched to when the first update could be sent in the recovered service. For comparison, we also measured the time required to start a fresh instance of the same service, with no logged state to recover. Figure 2 shows the results.

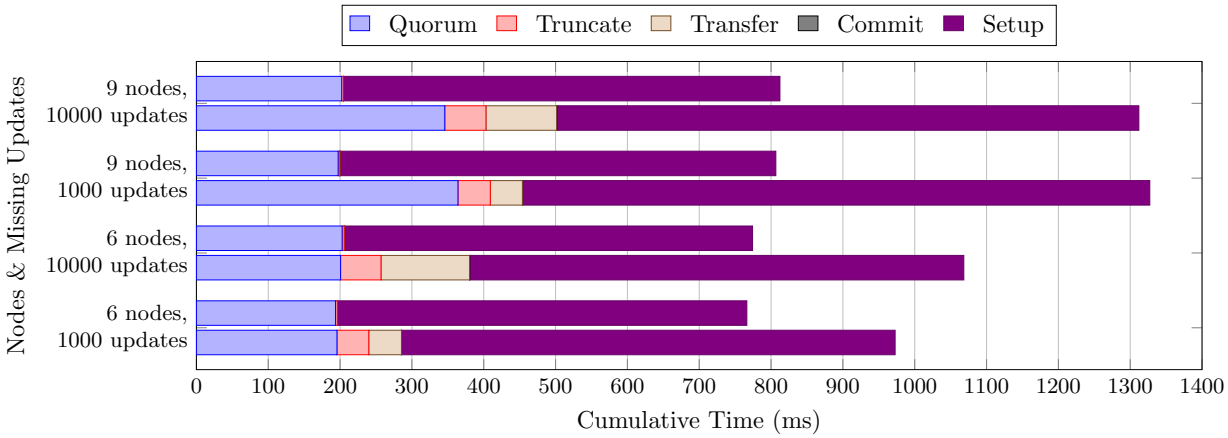


Fig. 4: Breakdown of time spent in each phase of starting or restarting a service, when 1 node per shard is out of date upon restart. Upper bars show fresh start, lower bars show restart.

We find that the restart algorithm adds only minimal overhead compared to the fresh-start case, and that the assignment of nodes into more or fewer shards does not have a noticeable effect on restart time, owing to the polynomial run time of min-cost flow. In both cases, the time to launch the service increases as the system scales up due to the fixed costs of initializing more distributed processes. For example, there is an increasingly variable delay in the time it takes each server to actually start the Derecho process after being given a command to do so.

Next, we measured the amount of metadata that was exchanged between the restart leader and the non-leader nodes in order to complete the restart algorithm, using the same setup as the experiment in Figure 2. (Metadata includes everything sent during the restart process except for the missing updates sent during state transfer). In Figure 3, we see that the restart leader sends and receives more metadata as the size of the overall group increases, increasing at an approximately linear rate. This is because the restart leader must contact every restarting node, both to receive its logged information and to send out the proposed restart view. However, the non-leader nodes exchange a nearly-constant amount of data regardless of the size of the group, since they only need to contact the leader and wait for its response. Note, also, that even at the largest group sizes, the leader only needs to receive a few kilobytes of data, aggregated over all of the restarting nodes.

In our next series of experiments, we evaluated the costs of restarting a system with one or more significantly out-of-date replicas (i.e. nodes whose logs are missing many committed updates). To do this, we created a Derecho service organized into shards of 3 nodes each, and allowed two out of three replicas in each shard to continue committing updates for some time after one replica had crashed. We then crashed the rest of the replicas, and restarted all of the nodes at once. Each update in this service contained 1KB of data.

Figure 4 shows a detailed breakdown of the amount of time spent in the four major phases of the restart algorithm in this situation: (1) awaiting quorum, (2) truncating logs to complete epoch termination, (3) transferring state to out-of-date nodes, and (4) waiting for the leader to commit a restart view. It also shows a fifth phase, which is the time spent in the setup process of the Derecho library before the first update can be sent; this includes operations such as pre-allocating buffers for RDMA multicasts. For comparison, we also measured the breakdown of time spent in a fresh start of the same service, which has only two phases: Awaiting quorum (i.e. waiting for all the processes to launch) and setting up the Derecho library.

This experiment shows even more clearly that our restart process is quite efficient compared to the normal costs of starting a distributed service. Even when one replica in each shard is missing 10000 committed updates, state transfer accounts for at most 120 ms, a small fraction of the overall time. It also shows the benefits of allowing each shard to complete state transfer in parallel: The 3-shard service spent no more time on state transfer than the 2-shard service, even though there were an additional 1000 or 10000 updates to send to an out-of-date node.

We also measured the number of bytes of data received by each out-of-date replica during the state-transfer process, and varied the amount of data contained in each update as well as the number of missing updates. The results are shown in Figure 5, and are fairly straightforward: the amount of data transferred to each out-of-date replica increases linearly with the size of an update, and with the number of updates that the out-of-date replica has missing from its log. Moreover, it is almost exactly equal to the number of missing updates multiplied by the size of each update, because the node did not need to download and merge logs from multiple other replicas. It is also important to note that this data is sent in parallel for each shard, so unlike the metadata in Figure 3, there is no difference in how much data any one node must send as the number of shards increases.

Finally, we measured the amount of time required to restart a service with out-of-date replicas as the size of each update scales up, shown in Figure 6. We found that for updates of sizes below 1 MB, neither the size of the update nor the

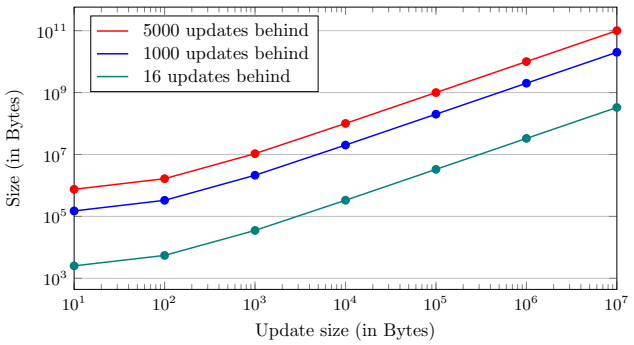


Fig. 5: Data downloaded by each out-of-date node, in a system with 3 shards of 3 members each.

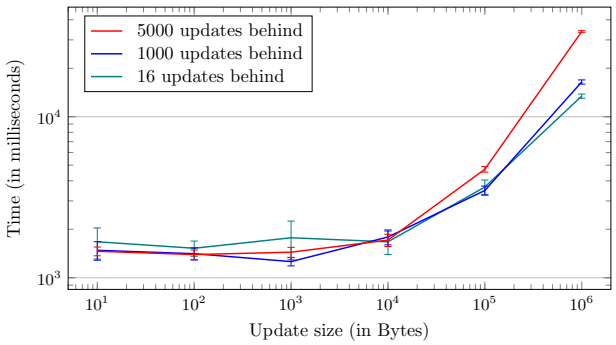


Fig. 6: Time to restart a service with 3 shards of 3 members each, with 1 out-of-date node per shard. Error bars represent 1 standard deviation.

number of missing updates on the out-of-date replicas had much of an effect on the restart time. For update sizes of 1MB and larger, the increasing amount of data that needed to be transferred to the out-of-date replicas had the expected effect of slowing down the restart process.

VI. Related Work

The algorithms implemented by Derecho combine ideas first explored in the Isis Toolkit [13, 6] with the Vertical Paxos model [14]. Other modern Paxos protocols include NOPaxos [4] and APUS [5]. Recent systems that offer a more durable form of Paxos, such as Spinnaker [15] and Gaios [16], include mechanisms for restarting failed nodes using their persistent logs. However, these papers generally do not consider the case in which every replica must be restarted at once. "Paxos Made Live" [17] explores a number of practical challenges (including durability) seen in larger SMR systems, a motivation shared by our work.

Bessani et al. looked at the efficiency of adding durability to SMR in [18], including the problem of minimizing state transfer during replica recovery. They provided a solution for recovering a non-sharded service in a Byzantine setting, and also showed how to lower the runtime overhead of logging and checkpointing. Their work did not look at services with complex substructure, which was a primary consideration here.

Corfu [9] is a recent implementation of SMR that uses a different approach from classic Paxos, distributing the command log across shards of storage-only nodes. Clients use Paxos to reserve a slot, then replicate data using a form of chain replication [19]. vCorfu [8] extends this by offering virtual sublogs on a per-application basis. However, if multiple subsystems use Corfu separately, recovery of the Corfu log might not recover the application as a whole into a consistent state. As we mentioned in sections II-A and IV-D, our protocol could be adapted to vCorfu to ensure that a quorum of replicas from each sublog of the last known layout is contacted before the system is restarted. Other replicated cloud services, such as Hadoop [20], Zookeeper [21], and Spark [22], employ an alternative approach to durability by ensuring that any state lost due to an unexpected failure can always be recomputed from its last checkpoint, but this is not an option in our setting.

Our work is inspired by a long history of distributed checkpointing and rollback-recovery protocols, many of which are summarized in [23], but updates these principles to the modern setting of replicated services and SMR. Rather than rely on an explicitly coordinated global checkpoint, as in [24] and [25], or attempt to record a dependency graph between locally-recorded checkpoints, as in [26], our system incorporates the dependency information already recorded in SMR updates to derive a globally consistent system snapshot from local logs.

Recovery of the final state of a single process group was first treated in Skeen's article "Determining the Last Process to Fail" [27]. Our scenario, with potentially overlapping subgroups, is more complex and introduces an issue of joint consistency they did not explore.

VII. Conclusion

Modern datacenter services are frequently complex, and may employ SMR mechanisms for self-managed configuration, membership management, and sharded data replication. In these services, application data will be spread over large numbers of logs, and recovery requires reconstruction of a valid and consistent state that preserves all committed updates. We showed how this problem can be solved even if further crashes occur during recovery, implemented our solution within Derecho, and evaluated the mechanism to show that it is highly efficient.

Acknowledgments

This work was supported, in part, by a grant from AFRL Wright-Patterson.

References

[1] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proc. 1996 ACM SIGMOD Int. Conf. Management of Data*. Montreal, Quebec, Canada: ACM, 1996, pp. 173–182.

[2] K. Birman, B. Hariharan, and C. De Sa, "Cloud-hosted intelligence for real-time IoT applications," *SIGOPS Oper. Syst. Rev.*, vol. 53, no. 1, pp. 7–13, Jul. 2019.

[3] S. Jha, J. Behrens, T. Gkountouvas, M. Milano, W. Song, E. Tremel, R. V. Renesse, S. Zink, and K. P. Birman, "Derecho: Fast state machine replication for cloud services," *ACM Trans. Comput. Syst.*, vol. 36, no. 2, pp. 4:1–4:49, Apr. 2019.

[4] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports, "Just say NO to Paxos overhead: Replacing consensus with network ordering," in *Proc. 12th USENIX Symp. Operating Systems Design and Implementation*. Savannah, GA, USA: USENIX Association, 2016.

[5] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui, "APUS: Fast and scalable Paxos on RDMA," in *Proc. 8th ACM Symp. Cloud Computing*. Santa Clara, CA, USA: ACM, Sep. 2017.

[6] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Trans. Comput. Syst.*, vol. 5, no. 1, pp. 47–76, Jan. 1987.

[7] L. Lamport, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[8] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritchie, S. Swanson, M. J. Freedman, and D. Malkhi, "vCorfu: A cloud-scale object store on a shared log," in *Proc. 14th USENIX Conf. Networked Systems Design and Implementation*. Boston, MA, USA: USENIX Association, 2017, pp. 35–49.

[9] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber, "CORFU: A distributed shared log," *ACM Trans. Comput. Syst.*, vol. 31, no. 4, pp. 10:1–10:24, Dec. 2013.

[10] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. 7th Symp. Operating Systems Design and Implementation*. Seattle, WA, USA: USENIX Association, 2006, pp. 307–320.

[11] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993.

[12] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, Feb. 1985.

[13] K. P. Birman, "Replication and fault-tolerance in the ISIS system," in *Proc. 10th ACM Symp. Operating Systems Principles*. Orcas Island, WA, USA: ACM, 1985, pp. 79–86.

[14] L. Lamport, D. Malkhi, and L. Zhou, "Vertical Paxos and primary-backup replication," in *Proc. 28th ACM Symp. Principles of Distributed Computing*. Calgary, Canada: ACM, Aug. 2009, pp. 312–313.

[15] J. Rao, E. J. Shekita, and S. Tata, "Using Paxos to build a scalable, consistent, and highly available datastore," *Proc. VLDB Endow.*, vol. 4, no. 4, pp. 243–254, Jan. 2011.

[16] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, "Paxos replicated state machines as the basis of a high-performance data store," in *Proc. 8th USENIX Conf. Networked Systems Design and Implementation*. Boston, MA, USA: USENIX Association, 2011, pp. 141–154.

[17] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," in *Proc. 26th ACM Symp. Principles of Distributed Computing*. Portland, OR, USA: ACM, 2007, pp. 398–407.

[18] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia, "On the efficiency of durable state machine replication," in *Proc. 2013 USENIX Annual Technical Conference*. San Jose, CA, USA: USENIX Association, Jun. 2013, pp. 169–180.

[19] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *Proc. 6th Symp. Operating Systems Design and Implementation*. San Francisco, CA, USA: USENIX Association, Dec. 2004, pp. 91–104.

[20] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. 26th IEEE Symp. Mass Storage Systems and Technologies*. IEEE Computer Society, 2010.

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for Internet-scale systems," in *Proc. 2010 USENIX Annual Technical Conference*. Boston, MA, USA: USENIX Association, 2010.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf. Hot Topics in Cloud Computing*. Boston, MA, USA: USENIX Association, Jun. 2010.

[23] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, Sep. 2002.

[24] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Trans. Softw. Eng.*, vol. SE-13, no. 1, pp. 23–31, Jan. 1987.

[25] E. N. Elnozahy and W. Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit," *IEEE Trans. Comput.*, vol. 41, no. 5, pp. 526–531, May 1992.

[26] B. Bhargava and S.-R. Lian, "Independent checkpointing and concurrent rollback for recovery in distributed systems – an optimistic approach," in *Proc. 7th Symp. Reliable Distributed Systems*, Oct. 1988, pp. 3–12.

[27] D. Skeen, "Determining the last process to fail," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 15–30, Feb. 1985.