ReCraft: Self-Contained Split, Merge, and Membership Change of Raft Protocol

Kezhi Xiong[1] Soonwon Moon[2] Joshua Kang[1] Bryant Curto[1] Jieung Kim[3] Ji-Yong Shin[1]
[1]*Northeastern University* [2]*Seoul National University* [3]*Yonsei University*

***Abstract*—Designing reconfiguration schemes for consensus protocols is challenging because subtle corner cases during reconfiguration could invalidate the correctness of the protocol. Thus, most systems that embed consensus protocols conservatively implement the reconfiguration and refrain from developing an efficient scheme. Existing implementations often stop the entire system during reconfiguration and rely on a centralized coordinator, which can become a single point of failure. We present ReCraft, a novel reconfiguration protocol for Raft, which supports multi- and single-cluster-level reconfigurations. ReCraft does not rely on external coordinators and blocks minimally. ReCraft enables the sharding of Raft clusters with split and merge reconfigurations and adds a membership change scheme that improves Raft. We prove the safety and liveness of ReCraft and demonstrate its efficiency through implementations in etcd.**

***Index Terms*—Consensus protocols, Reconfiguration, Sharding**

I. Introduction

Reconfiguration[1] of a distributed system is vital to keep the system scalable, cost-effective, and alive. For example, reorganizing the cluster depending on user loads can avoid service slowdowns or save operational costs, and timely replacement of failed nodes can preserve the system's availability.

The correctness of most distributed protocols relies on the configuration, and it is crucial that reconfigurations are performed in a consistent and fault-tolerant manner. Problems with reconfigurations can invalidate the core protocol assumption and stop the system [1]–[3]. Thus, many systems manage the configuration using strongly consistent and highly available services, such as Chubby [4], Zookeeper [5], and etcd [6].

Then how do these strongly consistent systems manage and change their configurations? Internally, these systems build on multi-Paxos [7]–[9], and Raft [10] which are consensus-based state machine replication (SMR) [11] protocols that are logically equivalent [12]. Reconfiguration schemes for consensus protocols can be categorized by whether they rely on an external configuration manager and whether the system must block during the reconfiguration process. Chubby [4] and vertical Paxos [13] assume an external configuration manager, which can be another multi-Paxos cluster. Most other Paxos variants manage the configuration within itself but require the reconfiguration command to be *committed* through consensus before execution [9], [14]–[19]. Raft improves them in a *wait-free* fashion which optimistically applies the new configuration right after the command is *received* [10], [20].

[1]Configurations can include various information, but our main interest is member nodes of a distributed system.

While reconfiguration schemes for multi-Paxos and Raft systems are capable of changing member nodes of a single cluster instance and they cannot fully address the multi-cluster reconfiguration that relates to scalability. Multi-Paxos and Raft can scale for the read throughput by adding passive learner nodes, but the write throughput does not scale by adding nodes. Consensus-based SMR protocols are designed to maintain identical copies of data typically through a single leader multicasting to all other members. Thus, adding more nodes only decreases the write throughput. Existing solutions to scale the write throughput include employing multiple leader nodes [21], logically partitioning physical nodes to concurrently commit different sets of commands [22], and decoupling log ordering and replication [23]. Still, these approaches focus on maintaining a single instance of the SMR cluster and do not support scaling out to many. Raft-based storage systems like TiKV [24] and CockroachDB [25] support scaling beyond a single cluster by managing multiple logical sharded instances of Raft. However, they require a coordinator module outside of Raft, which can be a single point of failure. Also, the protocol is designed conservatively to avoid potential bugs: cluster reconfigurations run for maximum two clusters at a time in sequential steps with frequent blocking.

In this paper, we present ReCraft, a reconfigurable Raft, which augments Raft with 1) a *self-contained* consensus-based concurrent cluster split and merge protocol that does not rely on any external service and 2) a new membership change protocol that is more fault-tolerant.

With ReCraft, a Raft instance can split into two or more instances that handle disjoint data sets, and multiple instances can merge into one. The split and merge decisions are made by the consensus of all participating nodes. The protocol mixes push and pull-based communication to ensure that disconnected nodes or clusters during the reconfiguration can catch up. The protocol looks simple, like the original Paxos or Raft protocol, which can be summarized in a few lines of pseudo code. However, ReCraft is a result of iterative revisions and formal proofs to handle subtle corner cases.

The ReCraft membership change alters multiple member nodes and quorum sizes in a single step in a wait-free fashion. The scheme relies on a recent formal proof of generalized Raft reconfiguration [26]. ReCraft instantiates the general theory into a practical multi-node reconfiguration design and implementation. Compared to the Raft joint consensus reconfiguration, it is simpler to reason and exhibits better fault tolerance for practical cluster sizes as it requires fewer votes.

To ensure that ReCraft protocols are bug-free and compatible with the main Raft protocol, we present proof of safety and liveness under the same assumption with Raft (Section VI). For safety properties, we prove that ReCraft preserves the immutability and linearizability of its state (i.e., state machine safety) under reconfigurations. For the liveness, we prove that the split and merge protocols keep all nodes and clusters alive. Due to the logical equivalence of Raft and multi-Paxos [27], [28], our protocol can also be ported to multi-Paxos.

We implement ReCraft in etcd [6] and its Raft library which many other systems, including TiKV and CockroachDB, rely on. Evaluations of ReCraft on a public cloud and a theoretical analysis against etcd/TiKV/CockroachDB demonstrate that ReCraft is safe, live, and efficient.

This paper makes the following contributions:

- To our knowledge, the first self-contained split/merge protocol for Raft and Paxos protocols.
- A new wait-free membership change protocol that improves Raft for practical cluster sizes.
- Formal safety and liveness proofs of ReCraft protocols.
- The implementation of ReCraft in etcd and an evaluation in a cloud environment.

II. Background

In this section, we introduce Raft [10] and its reconfiguration [20] which are the basis of ReCraft and discuss split/merge protocols for Raft in TiKV and CockroachDB.

A. Raft

1) Basic Workings

Raft replicates the SMR log using distributed consensus in a strongly consistent manner to a cluster of nodes in two stages: leader election and log replication.

Leader Election. Raft uses the monotonically increasing term and its SMR log records log entries with the term when the entry is generated. There is at most one elected leader per term. The leader constantly sends log replication requests or heartbeat messages to non-leader nodes or followers. Each follower maintains a random timer that resets if it hears from the leader. If the timer goes off, the follower turns into a candidate, increments the term number, and multicasts a leader election message for the term to all nodes in the cluster. The election message contains the term number and the recency of the candidate's local SMR log. Each node has one vote per term and sends its vote to the candidate who first contacts with the same or later log state than itself. The candidate that gathers the votes from a majority quorum of the cluster becomes the leader. If none of the candidates receives the majority votes, the leader election reoccurs in the next term.

Log Replication. All replication requests for log entries go through the leader. Similar to the leader election, the leader multicasts the log entry with the current term number to all nodes in the cluster. The followers append the entry only if the leader's term number is equal to or greater than the follower-perceived term and their logs match with the leader's log up to where the new log entry is about to be appended. In case a follower's log is out of sync, the leader first keeps the follower's log up-to-date and then appends the new log entry. If a majority quorum accepts the replication request, the log entry is committed and becomes immutable.

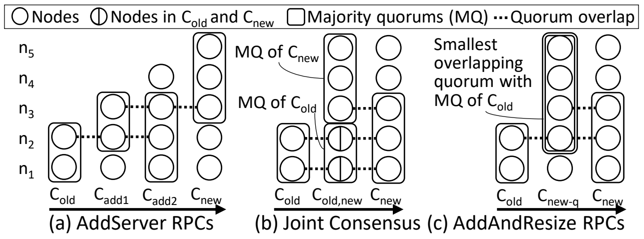


Fig. 1: Reconfiguring a 2-node cluster (C_{old}) to a 5-node cluster (C_{new}) using Raft and ReCraft reconfiguration schemes.

Quorum Overlap. The majority quorum support in each step ensures that the next leader is supported by at least one node that committed the last committed log entry and the elected leader's log contains all committed log entries. This quorum overlap is one of the key properties that lead to the safety guarantees of the protocol. Both Raft and ReCraft reconfiguration protocols heavily rely on the quorum overlap.

2) Membership Changes

Raft employs two types of membership change schemes: the Add/RemoveServer RPC (AR-RPC) for a single member node change in one consensus step [20] and the joint consensus (JC) for arbitrary node changes in two consensus steps [10]. Both are wait-free in which nodes optimistically apply the configuration change immediately after receiving it as a special SMR log entry and converge to the committed configuration.

Both approaches leverage the quorum overlap property between old and new configurations. The AR-RPC (Figure 1a) adds or removes one node at a time, and the one-node difference naturally maintains the quorum overlap: all majority quorums (MQ) of an old N node configuration C_{old} always overlap with those of new $N+1$ or $N-1$ node configuration C_{new}. On the other hand, the JC (Figure 1b) induces the quorum overlap and works in two steps. It starts with the old configuration C_{old} and enters the joint mode by committing the joint configuration $C_{old,new}$ to quorums of C_{old}. Under the joint mode, any decision should be made by the quorums of both old and new configurations, and these quorums subsume those of C_{old}. Then, the new configuration C_{new} is committed to a quorum of the new configuration, and the reconfiguration completes. Similarly, the quorums of $C_{old,new}$ subsume those of C_{new}, so the quorum overlap is always maintained.

B. Generalized Add/RemoveServer RPC

The single-step AR-RPC is simpler and faster than the JC but can only alter one node at a time. The restriction comes from preconditions to fulfill before starting the AR-RPC:

P1 All prior reconfiguration requests in the leader's log must be committed.

P2 The new configuration should differ from the current configuration by at most one node.

P3 The leader must commit a log entry in its term before starting a new reconfiguration.

Among them, **P3** was later added to deactivate concurrent reconfigurations that led to a subtle bug [29]. While formally verifying that **P3** indeed removes the bug, Honoré et al. [26] showed that **P2** can be relaxed to,

P2' Consecutive configurations should always maintain the quorum overlap.

They formally verified the safety of any reconfiguration scheme that satisfies the preconditions with this generalization, including the AR-RPC. The ReCraft membership change protocol is an instance that satisfies the safety proof and sits between the AR-RPC and the JC (Section IV).

C. Cluster Split/Merge in Multi-Raft Designs

TiKV and CockroachDB implement multi-Raft, which is multiple sharded clusters of Raft. Both systems employ similar splitting and merging mechanisms using an external cluster manager (CM) that drives the operation as follows [30].

Split. To split cluster C_{src}, the CM commits a new subrange command to C_{src}. The CM brings up cluster C_{dst} that will take the remaining subrange, and data is copied from C_{src} to C_{dst}. After the copying completes, C_{dst} starts the service.

Merge. To merge clusters C_{src} and C_{dst}, the CM stops C_{src} by committing a special command and copies data from C_{src} to C_{dst}. Once the data is fully copied, the CM increases C_{dst}'s range to include C_{src}'s range.

Split and merge operations of TiKV/CockroachDB can be viewed as the CM moving a subset of data and range from one cluster to a new cluster and migrating the entire data and range of a cluster to another cluster, respectively. While the centralized CM handling the split and merge simplifies the high-level process, interleaved or concurrent reconfigurations can complicate the operations. Thus, the CM uses distributed locks/transactions at the layers above to solve these problems.

III. Splitting and Merging ReCraft Clusters

Most production systems running consensus-based SMR implement key-value interfaces [5], [6], [31], and independent access to keys naturally lends the system to sharding as in distributed hash tables [32]–[35]. Based on our goal to design a consistent, failure-resilient, and live split and merge protocol for Raft clusters, we establish two design principles:

1. Rely minimally on external services or manual controls.
2. Utilize existing consensus mechanisms in Raft that guarantee strong consistency and high availability.

A. Preliminaries

Preconditions Splitting and merging reconfigurations of ReCraft are decided by the consensus of the member nodes similar to Raft. ReCraft applies the same preconditions **P1**, **P2'**, and **P3** for all reconfigurations to prevent outdated reconfiguration commands potentially coming from multiple clusters from interfering with the up-to-date one.

Failure Assumptions ReCraft shares the same failure model as Raft and Paxos: it assumes an asynchronous network and non-Byzantine node failures. Each ReCraft cluster, which we interchangeably refer to as a configuration C, can tolerate $f = n - q$ node failures, where n is the number of nodes in the cluster and q is the quorum size. q is typically the majority in Raft and Paxos, but during intermediate steps of ReCraft reconfigurations, q can temporarily grow larger than the majority but never smaller like Raft. ReCraft always guarantees safety (Theorem 1) under the failure model. However, for liveness, ReCraft, like Raft, assumes at least a quorum of nodes—which may vary depending on the reconfiguration phase—in each cluster reacts to requests within a finite number of retrials. Both for safety and liveness, any node (e.g., no exceptions for leader nodes) can fail at any time as long as these assumptions hold. ReCraft does not rely on external services except for a naming service in a rare case (Section V). Thus, the protocol is self-contained and there is no single point of failure even during reconfiguration, unlike other SMR systems with sharding like TiKV/CockroachDB [3], [24].

Epoch Numbers. We introduce monotonically increasing epoch numbers to order the configurations committed for splitting and merging. We add the epoch number as a prefix to Raft's term number (e.g., the first 4 bytes as the epoch number and the remainder as the regular term number for an 8-byte integer). The epoch numbers are located at higher digits of the new term number, so an updated epoch number for a split/merge will prevent commands from old configurations from interfering with the new configuration. Epoch numbers also work as indicators for nodes that failed to participate in the reconfiguration to realize that their peers have moved on. Note that epoch numbers are not updated for membership change reconfigurations within a single cluster.

B. Split

ReCraft can split a cluster into multiple subclusters[2]. They share the same SMR log of the original cluster but evolve independently after the split. The overlay layer, such as the Zookeeper or etcd, can selectively accept requests for the subcluster's new range or redirect them to appropriate subclusters.

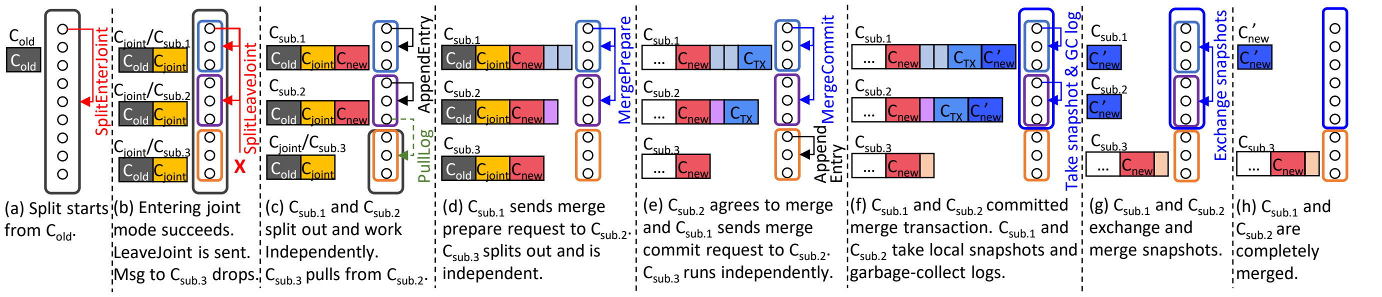
We propose a variant of the JC for split operations so that all subclusters have clues of the ongoing split even if they go offline in the middle. During the split, we use different election and log commit quorums. The pseudo code of our two-phased split protocol is in Figure 2 and an example is in Figure 3. Note that the return of "SUCCESS" in the code means the operation fully succeeded, but "FAILURE" means the operation may or may not have failed and requires a re-execution (e.g., a leader committing log entries from past terms in Raft).

Configurations. The new configuration C_{new} should include how many subclusters the current configuration C_{old} will split into and each new subcluster i's configuration $C_{sub.i}$. Each $C_{sub.i}$ includes disjoint members and data ranges of C_{old}. C_{joint} is the intermediate step to reach C_{new} and has the same information as C_{new}. C_{joint} only changes how the election

[2]We use the term subcluster to refer to clusters generated from a cluster split, or clusters merging to form a single cluster.

```
1  // Leader - enter joint consensus
2  bool SplitEnterJoint(Config $C_{joint}$) {
3    votes = [];
4    $C$ = getCurrentConfig();
5    // Precondition check
6    if (!P1 || !P2′ || !P3)
7      return FAILURE;
8    foreach (n in memberNodes($C$)) {
9      if (appendEntry(n, $C_{joint}$) == OK)
10       votes.add(n);
11   }
12   applyElectConfig($C_{joint}$);
13   if (isQuorum($C$, votes)) return
14     SUCCESS;
15   else return FAILURE;
16 }
17 // Leader - leave joint consensus
18 bool SplitLeaveJoint(Config $C_{new}$) {
19   votes = [];
20   $C$ = getCurrentConfig();
21   if (!isJoint($C$)||!isCommitted($C$))
22     return FAILURE;
23   foreach (n in memberNodes($C$)) {
24     if (appendEntry(n, $C_{new}$) == OK)
25       votes.add(n);
26   }
27   $C_{sub}$ = myConfig($C_{new}$);
28   applyCommitConfig($C_{sub}$);
29   if (isQuorum($C_{sub}$, votes)) {
30     notifyCommit($C_{old}$, $C_{new}$);
31     applyElectConfig($C_{sub}$);
32     IncEpoch();
33     return SUCCESS;
34   } else return FAILURE;
35 }
36 // Candidate
37 bool EnterElection() {
38   $C$ = getCurrentConfig();
39   foreach (n in memberNodes($C$)) {
40     res = requestVote(n)
41     if (res == OK) votes.add(n);
42     if (res == PULL) {
43       pullLog(n);
44       return FAILURE;
45   } }
46   if (isQuorum($C$, votes))
47     return SUCCESS;
48   else return FAILURE;
49 }
50 // Follower - response to request vote
51 void HandleVote(Request req) {
52   $E_{curr}$ = getCurrentEpoch();
53   $E_{req}$ = getEpoch(req.term);
54   if ($E_{curr}$ > $E_{req}$)
55     respondPull(req.node);
56   else ...  /*same as Raft voting*/
57 }
```

Fig. 2: Pseudo code of functions for the split protocol. Functions in red include communications: appendEntry, requestVote and PullLog are 1-to-1 RPCs, notifyCommit is a multicast RPC, and respondPull sends an acknoweldgment.



(a) Split starts from C_{old}.
(b) Entering joint mode succeeds. LeaveJoint is sent. Msg to $C_{sub.3}$ drops.
(c) $C_{sub.1}$ and $C_{sub.2}$ split out and work Independently. $C_{sub.3}$ pulls from $C_{sub.2}$.
(d) $C_{sub.1}$ sends merge prepare request to $C_{sub.2}$. $C_{sub.3}$ splits out and is independent.
(e) $C_{sub.2}$ agrees to merge and $C_{sub.1}$ sends merge commit request to $C_{sub.2}$. $C_{sub.3}$ runs independently.
(f) $C_{sub.1}$ and $C_{sub.2}$ committed merge transaction. $C_{sub.1}$ and $C_{sub.2}$ take local snapshots and garbage-collect logs.
(g) $C_{sub.1}$ and $C_{sub.2}$ exchange and merge snapshots.
(h) $C_{sub.1}$ and $C_{sub.2}$ are completely merged.

Fig. 3: An example of a series of split and merge operations from a cluster-level viewpoint. Rounded rectangles show clusters' node configurations and cluster's committed log states are on the left. C_{old} splits to $C_{sub.1-3}$ (steps a-d), and $C_{sub.1}$ and $C_{sub.2}$ merge to C'_{new} (steps d-h).

quorum works from C_{old}: majority quorum votes from each and every $C_{sub.i}$ is required. When C_{new} arrives to each node, the node extracts its own $C_{sub.i}$ from C_{new} and applies it.

SplitEnterJoint. To initiate a split, the leader proposes entering the joint mode with C_{joint} (Figure 3a). The preconditions **P1**, **P2'**, and **P3** must be satisfied (line 6) and C_{joint} is sent to the member nodes as a special log entry that applies immediately after *receiving*. The leader applies the configuration in a wait-free fashion regardless of the quorum acknowledgment (line 12). Applying the configuration only changes the election quorum and the quorum for log commit still uses C_{old}. Until C_{joint} is committed, the leader cannot propose to leave the joint mode, but it can propose regular log entries.

SplitLeaveJoint. To leave C_{joint}, the leader first confirms that C_{joint} is *committed* (line 21). Then it sends C_{new} to all members of C_{joint} (Figure 3b) and applies C_{sub} in a wait-free fashion (lines 23-28). However, C_{sub} is not fully applied at this stage: the leader communicates with nodes in C_{sub} for committing C_{new} and log entries that come after, communicates with all nodes in C_{old} to replicate C_{new} and entries that come before, and use the quorum of C_{joint} to elect a new leader, if necessary. After confirming the commit of C_{new}, the leader notifies the commit of C_{new} — similar to how a committed index of regular log entry gets notified in Raft — to all members of the C_{old} (line 29-30), fully applies C_{sub} (line 31), and updates the epoch number (line 32). From this point on, the split for C_{sub} of the leader is completed and C_{sub} can run independently of others.

The follower applies C_{sub} the same way: once it receives C_{new} it only changes the commit quorum, and the election quorum updates after confirming the commit of C_{new}. The commit notification (line 30) is necessary so that candidates from other subclusters, if they have C_{new} in their log, can know of its commit and elect a leader within its subcluster. If subclusters do not have the C_{new} log entry, they have to first catch up with the log (described below).

Pulling through EnterElection and HandleVote. In the event of a network disruption during the last phase of splitting, some nodes or even an entire subcluster could miss the message from SplitLeaveJoint. If the split succeeds for a C_{sub} (i.e., C_{new} is committed to the quorum of C_{sub}), nodes in C_{sub} that missed the message will be eventually contacted by its peers and catch up with the configuration. However, if an entire subcluster misses the message to leave the joint mode and other subclusters have moved forward, it has no way to make progress under a regular Raft-like setting. The members of the subcluster are in C_{joint} and they need quorum votes from all other subclusters to elect a leader. However, other subclusters will have more recent logs and higher epoch numbers, so they will not respond to the missed-out subcluster.

ReCraft adds a pull mechanism so that the missed-out subcluster ($C_{sub.3}$ in Figure 3b) can save itself. The missed-out subcluster must first learn about the split completion and it should catch up with the log status up to the entry of C_{new}. All ReCraft nodes can respond to the node that sends vote requests with a lower epoch number to pull log entries from themselves (line 55). The node receiving the pull response pulls the *committed* log entries from the responder (line 43 and

```
// Coordinator cluster leader
bool MergePrepare(Config $C_{TX}$) {
  2pcVotes = []; votes = [];
  $C$ = getCurrentConfig();
  // Precondition check
  if (!P1 || !P2' || !P3)
    return FAILURE;
  // Local TX decision to own cluster
  $C_{TX}'$ = setDecision($C_{TX}$, OK);
  votes = appendEntryToConf($C$, $C_{TX}'$);
  // send 2PC prepare to $C_{sub}$
  foreach ($C_{sub}$ in clusters($C_{TX}$)) {
    if (Prepare2PC($C_{sub}$, $C_{TX}$) == OK)
      2pcVotes.add($C_{sub}$);
  }
  if (isQuorum($C$, votes) &&
      isUnanimous($C_{TX}$, 2pcVotes)) {
    applyConfig($C_{TX}'$);
    return SUCCESS;
  } else return FAILURE;
}
// Non-coordinator cluster leader
bool HandleMergePrepare(Config $C_{TX}$) {
  votes = [];
  $C$ = getCurrentConfig();
  // Precondition check
  if (!P1 || !P2' || !P3) res = NO;
  else res = OK;
  // Local TX decision to own cluster
  $C_{TX}'$ = setDecision($C_{TX}$, res);
  votes = appendEntryToConf($C$, $C_{TX}'$);
  if (isQuorum($C$, votes)) {
    applyConfig($C_{TX}'$);
    respond(res);
    return SUCCESS;
  } else return FAILURE;
}
// Coordinator cluster leader
bool MergeCommit(bool res, $C_{new}$) {
  2pcAcks = [];
  votes = [];
  $C$ = getCurrentConfig();
  if (!underTX($C$)||!TXPrepared($C_{TX}$))
    return FAILURE;
  // Final TX decision to own cluster
  $C_{next}$ = (res == COMMIT)?$C_{new}$:$C_{abort}$;
  votes = appendEntryToConf($C$, $C_{next}$);
  // send 2PC commit/abort to $C_{sub}$
  foreach ($C_{sub}$ in clusters($C_{TX}$)) {
    if (Commit2PC($C_{sub}$, $C_{next}$) == OK)
      2pcAcks.add($C_{sub}$);
  }
  if (isQuorum($C$, votes) &&
      allResponded($C_{TX}$, 2pcAcks)) {
    applyConfig($C_{next}$);
    return SUCCESS;
  } else return FAILURE;
}
// Non-coordinator cluster leader
bool HandleMergeCommit(Config $C_{next}$) {
  votes = [];
  $C$ = getCurrentConfig();
  votes = appendEntryToConf($C$, $C_{next}$);
  if (isQuorum($C$, votes)) {
    applyConfig($C_{next}$);
    respond(OK);
    return SUCCESS;
  } else return FAILURE;
}
```

Fig. 4: Pseudo code of functions for the two-phase commit of the merge protocol. Functions in red include communications: Prepare2PC, and Commit2PC are 1-to-1 RPCs, appendEntryToConf is a multicast RPC, and respond sends an acknowledgement.

Figure 3c). This is similar to the routine log-catchup operations during the appendEntry RPC of Raft in reverse directions which does not incur high-overhead. Once the C_{new} entry is pulled, the subcluster applies C_{sub} and elects its leader.

Epoch numbers play two critical roles here: 1) they prevent missed-out nodes of other subclusters from turning leaders of up-to-date subclusters into followers with a large term number, and 2) they clearly flag the commit of C_{new} so that the missed-out nodes can confidently pull log entries. Uncommitted log entries can be overwritten so a node in C_{new} halfway (i.e., applied C_{new} as the commit configuration but not election configuration) should not respond to pull requests. However, without the epoch number, whether the node has fully moved on to C_{new} or not is unclear from external observers' point of view. The epoch number clearly marks the commit of C_{new} and from which nodes shoiuld the missed-out nodes pull.

Differences from the Raft Joint Consensus. Unlike Raft's joint configuration $C_{old,new}$, ReCraft's C_{joint} does not explicitly require quorums from C_{old} and C_{new} for election and commit. C_{joint} only require quorums of all $C_{sub.i}$ for election as they are guaranteed to overlap with the quorum of C_{old}.

Using different quorums for leader election and log commit is also very different from Raft. As quorums of C_{old} and C_{joint} overlap, it is safe to commit to C_{old} under the joint mode. The quorum of C_{old} is equal to or smaller than that of C_{joint} so this expedites commit. Using the quorums of C_{joint} for election until completely committing C_{new} ensures all leaders elected until the split completes share the same view of the log up to C_{new} entry; immediately reducing the election quorum size can violate quorum overlap properties for log entries between C_{joint} and C_{new} and break safety guarantees. The use of different quorum sizes for election and commit resembles Flexible Paxos [36], but ReCraft uses this idea primarily to guarantee safety among concurrent subclusters.

Pulling log entries from other subclusters is a fully new feature in ReCraft. It resembles a Paxos-style leader election where proposers collect up-to-date data from other nodes, or learner nodes that collect committed states from others.

Subtle Corner Cases. During the pull operations, the data source node may not have all the necessary log entries or may even have a more outdated log than the puller. If the source node was also a missed-out node and only received a request vote message from an up-to-date node with the updated epoch number, then its term/epoch number will become up-to-date but its log state could still be outdated. Since the pull operation only copies committed log entries, it does not break safety. The puller can contact different nodes for the latest data or wait for the outdated node to be updated.

Epoch numbers are updated only after confirming the commit of C_{new} instead of when partially applying C_{new} after receiving it. We initially thought of fully applying C_{new} and increasing the epoch number with the configuration. However, 1) as explained above, the safety can break for immediately changing the election quorum size, and 2) it made the epoch number grow arbitrarily large for a single split if C_{new} fails to commit multiple times; this confuses the nodes to needlessly pull data from each other. The current design ensures safety and clearly marks a split success with the epoch increase.

C. *Merge*

The ReCraft merge protocol consolidates multiple clusters that manage disjoint data into one. The merge is more involved than the split, as it synchronizes and unifies states of multiple subclusters. The merge protocol consists of three phases (Figure 3d-h): the first two phases match each step of the two-phase commit (2PC) protocol to make the merge decision as a transaction, and the third phase exchanges subcluster states to create a common ground to resume as a single cluster. The data exchange phase blocks because the data transfer takes time, and at least a quorum of nodes in the merged cluster must be in the same state to preserve safety properties for leader election and log commits. Unlike the split that extends the JC, the merge adds new mechanisms to Raft.

Configurations and Overview. The new configuration of the merger C_{new} holds the members of the resulting cluster and from which subcluster $C_{sub.i}$ they come. C_{new} also includes the combined data range, a unique transaction id, and which subcluster is coordinating the merger. To reach C_{new}, the subclusters must go through the 2PC, where the transaction decision depends on all merging subclusters $C_{sub.i}$'s. As the first phase of the 2PC, all subclusters receive C_{TX}, which includes C_{new} as the transaction intent. Individual subclusters decide whether to join the merger or not and commit C_{TX} with the local decision. As the second phase of the 2PC, if all clusters agree to merge, then C_{new} is committed to all clusters; otherwise, C_{abort}, which nullifies C_{TX}, is committed. Between committing C_{TX} and C_{new}/C_{abort}, each cluster behaves the same, servicing regular client requests of the original configuration. Once C_{new} is committed, the data exchange happens among the subclusters. Finally, the merged cluster elects a new leader and resumes as a single cluster.

1) Distributed Transactions for Merge Decisions

The 2PC protocol for the merge runs at the cluster level (pseudo code in Figure 4), and it begins when the merge request arrives at one of the merging subclusters. The entire subcluster that was contacted becomes the coordinator to drive the 2PC. All ReCraft nodes have the logic to drive the 2PC and the leader of the subcluster takes the lead: the coordinator is naturally as robust as the Raft cluster unlike TiKV/CockroachDB's cluster manager.

MergePrepare. The leader of the coordinator subcluster starts the merger by triggering MergePrepare (Figure 3d). Similar to the split, it checks the three preconditions for reconfiguration (lines 6) and commits the local "OK" decision to its own cluster (lines 7-10). Then, it sends the 2PC prepare request to all involved clusters (lines 11-15). If the local commit succeeds and all clusters acknowledge "OK" then the MergePrepare succeeds (lines 16-19) and can move on to committing C_{new}. If any of the subclusters acknowledged "NO," then the merger fails, and the next step is to commit C_{abort}.

HandleMergePrepare. The non-coordinator subclusters treat the 2PC prepare request similarly to a regular client request and the leader of the subcluster handles the request with HandleMergePrepare. The main reason for the "NO" vote is **P1**, an ongoing reconfiguration, as **P2'** is guaranteed by the protocol and **P3** can be easily fulfilled by committing a no-op log entry (lines 27-28). Even when the cluster votes "NO," the decision must be recorded for proper execution of the 2PC protocol (lines 30-31). The response to the coordinator subcluster is sent after the decision is committed (lines 32-36).

MergeCommit and HandleMergeCommit. Once the transaction decision is finalized, the commit/abort phase of the 2PC takes place (Figure 3e). If the decision is an "ABORT" then the coordinator sends C_{abort} to all subclusters; otherwise, C_{new} is sent (lines 45-52). Reconfigurations C_{abort}/C_{new} must be applied after it is *committed* to the local subcluster (lines 60-65). This is required, because nodes successfully applying C_{new} immediately move on to the data exchange phase. The coordinator cluster applies the configuration last after checking that all subclusters completed the 2PC (lines 53-56).

Handling Failures. The merge issues subcluster-granularity operations and as long as each subcluster is alive (i.e., satisfies our failure assumption for a cluster), the merge operation goes through. For example, a complicated failure scenario is the leader node of the coordinating cluster failing. In this case, recovery schemes for Raft and 2PC are used (not in Figure 4). With the Raft recovery, the log entries related to the ongoing transaction will be noticed and committed to the local subcluster, if necessary, by the new leader. Then, the new leader will learn from the committed logs which stage of the 2PC execution was interrupted. 2PC transactions are designed to be idempotent using unique ids, and the new leader can resume the 2PC from the last known successful state.

2) Data Exchange and Resumption.

After the merge decision is finalized, the members of the merged cluster should resume from the same SMR log state.

Data Exchange. There can be many ways to merge subcluster states, such as concatenating or merge-sorting the logs. However, we choose to take snapshots of the replayed log (i.e., the key-value map of the etcd layer), exchange them, and use the combined snapshot as the base state for the merged cluster. We adopt the snapshot exchange because the snapshot is generally smaller than the log (e.g., the log can contain multiple updates to the same key but the snapshot of the replayed log only remembers the last update).

Once C_{new} is committed, the snapshot exchange takes place. Applying C_{new} includes creating a snapshot of the local log up to the log entry before C_{new}, trimming the log, and pulling the snapshots from other subclusters to create the combined snapshot (i.e., a key-value store state with combined key ranges). With the snapshot, nodes in the merged cluster start fresh with the log that begins with the C_{new} entry (Figure 3f-g). Note that log entries in subclusters that come after the C_{new} entry are discarded; logs are committed in a sequence, so it is safe to discard these uncommitted entries.

The basic ReCraft consistency guarantee is linearizability (or strict serializability of key-value pairs in the etcd context) on each cluster. However, the merge combines multiple linearized data chunks that are partially ordered. Thus, we regard the merged cluster—which inherits the ordering of each subcluster—as the start of a new cluster where linearizable updates begin; this is the same as TiKV and CockroachDB. While the current implementation only deals with disjoint data chunks, if the application (e.g., the etcd layer) running on ReCraft requires special ordering guarantees across merging data, additional logic can be implemented in the application.

Resumption. Before resuming operations, the merged cluster must decide the new epoch and term numbers to use. Before the commit of C_{new} and the exchange of snapshots, each subcluster uses its own epoch and term numbers. During the first phase of the 2PC, the coordinator collects the epoch numbers of all subclusters, and includes a new epoch number E_{new} greater than the maximum E_{max} of all subclusters in C_{new}: i.e., $E_{new} = E_{max} + 1$. After C_{new} is applied, C_{new} entry is treated as committed at term 0 of epoch E_{new}.

The first leader election of the merged cluster starts at term 1 of epoch E_{new}. However, not all nodes of C_{new} may be in sync when the first leader election occurs. Still, it is guaranteed that all subclusters have committed C_{new}: a candidate with E_{new} can arise only after merging the snapshots from all subclusters, and exchangeable snapshots become available after committing C_{new}. In a sense, the merge is a reverse process of the split where the 2PC mimics the joint consensus mode in which every subcluster has to approve the merge decision. For the merged cluster to carry on in C_{new} with data from all subclusters, it is sufficient to use the majority quorum. Note that nodes that are unaware of C_{new} cannot become the leader due to the small epoch number and outdated log.

Handling Missed-out Nodes. The merge protocol only requires a majority quorum from each subcluster to complete the merger, and there can be nodes that are behind. Unlike the split, the up-to-date log of the merged cluster looks very different from that of old subclusters, since the log is garbage collected and the cluster state relies on the merged snapshot.

ReCraft employs two ways to keep the nodes up-to-date. First, the missed-out nodes can update themselves by contacting other nodes that have higher epoch numbers using EnterElection and pullLog functions that are used in the split process. Second, the leader node that knows the entire reconfiguration history and monitors the follower's log status during appendEntry RPC call can install the snapshot and missing log entries. This is similar to Raft's InstallSnapshot RPC [20] but works for nodes coming from different subclusters.

Resizing the Merged Cluster. The merged cluster naturally holds a large number of member nodes, and it is likely that the cluster size needs to be reduced. The membership change reconfiguration after the merge works, but the resizing can be done while applying C_{new}. The ReCraft merge protocol resembles the stoppable Paxos [14], where a cluster temporarily stops for the reconfiguration. Thus, it is possible to select a subset of nodes in C_{new} to resume as the merged cluster. However, an arbitrary node selection can lead to choosing only missed-out nodes that are outdated. The safety requirement is to guarantee quorum overlap between C_{new} and the resized cluster: selecting all members of one or more C_{sub} fulfills this. Such member information can be optionally added to C_{new}.

IV. ReCraft Membership Change

The membership change of a single cluster is another important reconfiguration for the scalability and longevity of a cluster. We propose a variant of the AR-RPC and the JC for the single cluster membership change based on the generalized proof surrounding **P2'** [26] (Section II-B). Similar to vanilla Raft reconfigurations, the proposed scheme is wait-free and always maintains quorum overlaps between consecutive configurations. ReCraft can add/remove multiple nodes to/from a cluster at once, but unlike the JC that requires votes from a specific set of nodes (i.e., from old and new configurations), ReCraft does not distinguish the source.

A. Add/RemoveAndResize and ResizeQuorum

Adding Nodes. ReCraft offers AddAndResize RPC to add n new nodes to configuration C_{old} that has N_{old} members. The RPC adds n nodes to the cluster, modifies the quorum size from Q_{old} to $Q_{new-q} = N_{old} + n - Q_{old} + 1$, and places the cluster into an intermediate configuration C_{new-q} in a single consensus step (Figure 1c). "New-q" indicates that the configuration has the same members as the final configuration C_{new} but uses a different quorum size Q_{new-q} for both leader election and log commit. Q_{new-q} is the smallest quorum size that forces quorums of C_{old} and C_{new-q} to overlap.

The RPC adds n nodes to C_{old} in a single step, but the quorum size Q_{new-q} needs to be adjusted to the majority for the cluster to work like a regular Raft cluster. To reach this final configuration C_{new}, the ResizeQuorum RPC is used to reset the quorum size to $Q_{new} = \lfloor \frac{N_{new}}{2} \rfloor + 1$ in a single consensus step. C_{new-q} and C_{new} share the same members, and Q_{new-q} is always equal to or greater than Q_{new}, so the quorums of C_{new-q} and C_{new} are guaranteed to overlap. Thus, the safety of ResizeQuorum RPC holds.

Removing nodes. Removing nodes from the cluster works similarly using the RemoveAndResize RPC to reach C_{new-q} and then calling ResizeQuorum RPC to reach the target C_{new} with the majority quorum. The quorum size in C_{new-q} during the removal is adjusted to $Q_{new-q} = N_{old} - Q_{new} + 1$ to force the quorum overlap between C_{old} and C_{new-q}.

Unlike the AddAndResize RPC, which can add an unbounded number of nodes, the RemoveAndResize RPC can remove up to $r < Q_{old}$ nodes from C_{old}. Because the maximum quorum size of C_{new-q} is capped by the cluster size N_{new-q}, if r becomes equal to or greater than Q_{old}, the quorum overlap between C_{old} and C_{new-q} cannot be satisfied.

B. Comparisons with Raft's Membership Changes

Number of Consensus Steps. When adding or removing one node, the AR-RPC performs better than the JC, as the AR-RPC requires one consensus step. ReCraft works the same as the AR-RPC as one node difference makes Q_{new-q} and Q_{new} to be equal and obviates the ResizeQuorum RPC. When adding or removing two nodes, ReCraft performs the best. The AR-RPC must be called twice to add/remove two nodes, which requires the same two consensus steps as the JC. ReCraft can handle adding two nodes in a single step when C_{old} has an even number of nodes and Q_{new-q} equals Q_{new}. Otherwise, ReCraft takes the same two consensus steps as the JC. Beyond altering two nodes, ReCraft takes the same number of consensus steps as the JC. However, if the cluster size needs to be reduced by more than half, which is rare in practice, then ReCraft requires extra steps.

Number of Necessary Vote Messages and Fault Tolerance. Beyond the number of consensus steps, how do ReCraft and the JC differ? The main difference is the number of votes to collect to commit a log entry in the intermediate configurations. ReCraft needs votes from a quorum larger than the majority (i.e., Q_{new-q}) and the JC requires majority quorum votes from both old and new configurations. For example, in Figure 1, ReCraft always needs votes from at least four nodes in C_{new-q}, but the JC under $C_{old,new}$ requires two votes from C_{old}= $\{n_1, n_2\}$ and three votes from

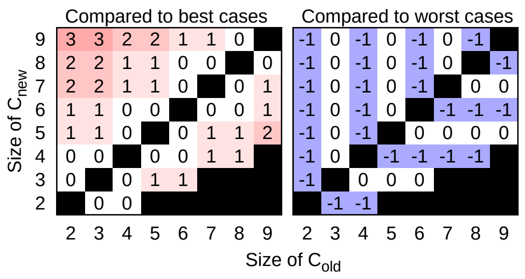


Fig. 5: The number of additional votes ReCraft requires during intermediate steps of the membership change compared to the best cases and the worst cases of the JC.

$C_{new}=\{n_1, n_2, n_3, n_4, n_5\}$. Depending on which order the votes arrive to the leader, the JC can commit a log entry with three to five votes. n_1 and n_2 belong to both C_{old} and C_{new}, so their votes count twice for each configuration. If their votes arrive first, one additional vote can fulfill the joint quorum. However, if votes of n_3, n_4, and n_5 arrive first, and the leader should wait for votes of n_1 and n_2 to fulfill the quorum of C_{old}. For adding and removing nodes, the number of necessary votes for the best case is $V_{best} = max(Q_{new}, Q_{old})$, and the worst case is $V_{worst} = |N_{new} - N_{old}| + min(Q_{new}, Q_{old})$.

Necessary vote counts relate to fault tolerance and tail latency if there are stragglers. In Figure 1, ReCraft can tolerate any one node failure under C_{new-q}. The JC under $C_{old,new}$ can tolerate two node failures out of $\{n_3, n_4, n_5\}$, which is better than ReCraft, but it cannot tolerate any failure of $\{n_1, n_2\}$, which makes ReCraft better. In essence, ReCraft conservatively sets the quorum size for the quorum overlap to treat all nodes equally. Still, ReCraft requires the same or fewer votes than V_{worst} of the JC.

The cells in Figure 5 show how many more votes ReCraft needs than the JC when reconfiguring a cluster of size on the x-axis to a new size on the y-axis. Positive numbers in red mean that the JC requires fewer votes, and negative numbers in blue mean ReCraft requires fewer votes. Compared to the best cases for the JC (left) when votes for the nodes in both old and new configurations arrive first, the JC always requires fewer votes than ReCraft when adding or removing more than two nodes. However, for more practical cases of adding or removing one or two nodes, ReCraft and JC perform comparably. ReCraft requires the same number of votes for altering one node and the same or one more votes for altering two nodes. Compared to the worst cases for the JC (right), ReCraft always requires the same or fewer votes.

Overall, the required votes for ReCraft and the JC are comparable, but for configuring member nodes between the most widely used Raft cluster sizes of 3 and 5, ReCraft is a more appealing choice for the number of consensus steps as the fault tolerance is comparable. Additionally, ReCraft's membership change protocol is simpler to reason and program as it treats all nodes equally.

V. Recovery from Long-Term Failures

Continuous split, merge, and membership changes can confuse nodes that were partitioned for an extended period. Following our first design principle of relying minimally on external parties, ReCraft employs means to restore these nodes.

Restoring a Node. Restoring the offline node when it gets back online is simple. When any of the live, up-to-date nodes perceive this node as their cluster member, they will eventually contact and update the node. Alternatively, if the peer nodes in the disconnected node are still online, the node can contact them to pull up-to-date log entries from them and catch up.

Restoring a Cluster. Restoring a long-term offline cluster is more subtle. There could be cases where the cluster missed SplitLeaveJoint call during a cluster split. If the peer clusters are around, the pull-based recovery approach in Section III-B suffices. However, 1) all peer clusters could have merged with other clusters, garbage collected the logs, and no longer have records that the offline cluster needs, or more radically 2) all members of the peer clusters could have been removed.

To handle the first case, ReCraft requires all clusters to maintain the reconfiguration history even after garbage collecting the log. Thus, even if clusters have gone through a series of reconfigurations, the offline cluster can restore itself from nodes that were in the same configuration. The reconfiguration history is garbage collected after confirming that all nodes involved have completed the reconfiguration.

For the second case, ReCraft uses a naming service that maintains the information of all live clusters. All clusters register their configuration to the naming service after reconfigurations. The naming service needs to be consistent with the cluster with a very loose time bound like the domain name service (DNS). Through the service, the cluster can find nodes that share the reconfiguration history, pull the history, and restore itself. This is the only time ReCraft relies on an external service.

Note that in a practical deployment, distributed systems commonly leverage a DNS-like naming service so that clients can find individual systems and bootstrap (e.g., Chubby, Google's etcd-like service relies on the naming service [4]). Similarly, ReCraft relies on this service for liveness, but we can instead make successor nodes of old configurations to monitor and restore all left-out nodes. Although this would implement a truly self-contained protocol, it adds unnecessary complexity to handle a very rare failure scenario.

VI. ReCraft Safety and Liveness Proofs

To guarantee the safety and liveness of ReCraft, we offer relevant proofs for split and merge. We only describe the main components of the proof and the remaining proof is available in the Appendix. We omit the proof for membership changes, as the proof by Honoré et al. [26] subsumes them.

A. Safety

The safety proof for ReCraft follows a similar structure to existing Raft protocol proofs [37], [38], and the top-level theorem is stated as:

Theorem 1 (State machine safety): If a node has applied a log entry at a given index to its state machine, no other node will ever apply a different log entry for the same index.

To prove Theorem 1, we start by defining analogous properties of ReCraft to those of Raft (Definitions 1-4) and adjusting sub-lemmas to introduce epoch numbers and multiple clusters.

Definition 1 (Leader Append-Only): A leader never overwrites or deletes its log entries and only appends.
Definition 2 (Election Safety): At most one leader can be elected *in a given epoch and term per cluster*.
Definition 3 (Log Matching): If two nodes of *the same epoch and cluster configuration* contain an entry with the same index and term in their log, then the logs of that epoch are identical in all entries up through the given index.
Definition 4 (Leader Completeness): Given an epoch number e and a cluster C, if an entry is directly committed in a term of (C, e), then that entry will be present in the logs of the leaders for all higher-numbered terms of (C, e).

We also introduce a new definition of consensus and quorums to distinguish different kinds of consensus in ReCraft:

Definition 5 (Consensuses and quorums): There are three kinds of consensuses used in ReCraft.

- *Normal consensus* for a cluster C requires its majority.
- *Joint consensus* for a set of subclusters $C_1, \ldots, C_n$ requires majorities for each $C_1, \ldots, C_n$.
- *Constituent consensus* for a set of clusters $C_1, \ldots, C_n$ requires a majority for one of $C_1, \ldots, C_n$.

A quorum for a consensus c is a specific set of nodes constituting the consensus.

Demonstrating that ReCraft satisfies analogous lemmas to those of Raft is largely straightforward by following Raft's proof, though it necessitates modifications due to changes in ReCraft. For instance, split and merge operations introduce new constraints beyond those relied upon by the original Raft:

1) A C_{new} entry cannot be appended to the log until the C_{joint} entry is committed.
2) The candidate holding the C_{joint} entry in its log should use C_{joint} for its election.
3) The leader under C_{joint}/C_{TX} cannot propose a new configuration other than the corresponding C_{new}/C_{abort}.

These modifications require defining properties related to the cluster reconfiguration of ReCraft, as in Definition 6. It asserts that any cluster split or merge will yield properly formed clusters ensuring the safety of our cluster configuration.

Definition 6 (Cluster Well-Formedness): Given an epoch number e and given two clusters C_1 and C_2 of epoch e, C_1 and C_2 are either identical or disjoint.

Another key characteristic of ReCraft is relevant to log updates as in Definition 7. It ensures that all committed entries are unique in terms of their index, epoch, and cluster.

Definition 7 (Log Consistency): Given an epoch number e and a cluster C, if two entries are committed at the same index in any member of (C, e), then they are indeed equal.

Reflecting the modifications in ReCraft, we design our proofs in three levels. Instead of proving all properties directly through induction on epoch and term numbers, we focused on,

1) Directly proving the induction on the step execution of ReCraft for properties independent of the increase in term and epoch numbers (e.g, Definition 1).
2) Demonstrating sub-lemmas (e.g., Definitions 2, 3, and 4) within a single epoch number based on the induction of term numbers. The key strategy for proving these properties is to establish that Definition 6 always holds and then prove the aforementioned sub-lemmas as corollaries.
3) The top-level safety theorem (Theorem 1) can be derived from the corollary lemma (Definition 7), which can also be proven by combining the stated sub-lemmas.

We focus on describing proof strategies for Definition 4, one of the most non-obvious theorems, rather than outlining all property proofs. It significantly alters original Raft's definition due to the introduction of cluster and epoch. To simplify the proof, we leverage Definition 6 to show that all the cooperating nodes of that epoch share the same cluster configuration.

For example, consider a situation where a follower a of epoch e and cluster configuration C_a accepted a message from a leader p with cluster configuration C_p. Then a and p share the epoch number since otherwise a wouldn't have accepted the message. Moreover, C_p contains a as its member since a leader can't send a message to a non-member. Then C_a and C_p have at least one common element, a. Then by the Cluster Well-Formedness, we can conclude that $C_a = C_p$. With this in mind, we prove Lemma 1.

Lemma 1 (Leader Completeness): Given an epoch number e, the Cluster Well-Formedness of epoch e (Definition 6) implies the Leader Completeness of epoch e (Definition 4.)
Proof 1: By assuming the Definition 6 of epoch e, we ignore any concerns about cluster configuration while proving Lemma 1. In the following proof, we assume all nodes are members of the same cluster (C, e).
Suppose a log entry $i_0 \mapsto (x_0, t_0)$ is directly committed by a leader p_0 at term t_0 and a leader p_1 is elected at t_1 where $t_0 < t_1$. Let log_0 be the log of p_0 at the time of committing $i_0 \mapsto (x_0, t_0)$, and log_1 be the log of p_1 at the time of its election. We have to show that $i_0 \mapsto (x_0, t_0)$ is contained in log_1. We use lexicographic strong induction on (i_0, t_1).
Let (c_0, q_0) be the consensus and the quorum used for committing $i_0 \mapsto (x_0, t_0)$ and (c_1, q_1) be the consensus and the quorum used for the leader election of $p_1@t_1$. c_0 can be one of *normal* or *constituent consensus*, while c_1 can be one of *normal* or *joint consensus*. We claim that q_0 and q_1 are not disjoint. Use case analysis on c_0.
Case 1: If c_0 is a *normal consensus*, we can immediately show that q_0 and q_1 are not disjoint.
Case 2: If c_0 is a *constituent consensus*, it means that there

is a C_{new} entry in log_0. In this case, there must exist a leader who created the C_{new} entry. With this knowledge, there must be a committed C_{joint} entry at the time of creating the C_{new} entry. Consider a directly committed entry $i' \mapsto (x', t')$ which leads to the commit of the C_{joint} entry. In this case, $i' < i_0$. Therefore, by the induction hypothesis with $i' \mapsto (x', t')$ and $p_1@t_1$, it can be derived that log_1 contains $i' \mapsto (x', t')$. By Definition 3, log_1 also contains the C_{joint} message. Hence, $p_1@t_1$ must use joint consensus for i ts election: i.e., c_1 is a *joint consensus*. Thus, we conclude that q_0 and q_1 are not disjoint.

Let v be a node of $q_0 \cap q_1$ and log_v be the log of v at the time of voting to p_1. We claim that log_v contains $i_0 \mapsto (x_0, t_0)$. By using the induction hypothesis with $i_0 \mapsto (x_0, t_0)$ and leaders for the term between t_0 and t_1, we know that all the intervening leaders contained $i_0 \mapsto (x_0, t_0)$. v accepted $i_0 \mapsto (x_0, t_0)$ at term t_0 so it also had the entry. Since followers only remove entries if they conflict with the leader, we can conclude that $i_0 \mapsto (x_0, t_0)$ would have preserved until v's vote to p_1. Since v granted its vote to $p_1@t_1$, there are two cases to consider.

Case 1: When $\text{lastTerm}(log_v) < \text{lastTerm}(log_1)$, assume that $i' \mapsto (x', t')$ is the last entry of log_1. Since $t_0 \leq \text{lastTerm}(log_v) < \text{lastTerm}(log_1) = t'$, we can use the induction hypothesis with $i_0 \mapsto (x_0, t_0)$ and the leader of t'. Then the leader of t' must have contained $i_0 \mapsto (x_0, t_0)$ at the time of election. By Definition 1, the log of the leader of t' at the time of creating $i' \mapsto (x', t')$ also contains $i_0 \mapsto (x_0, t_0)$. Then, according to Definition 3, we can conclude that log_1 contains the entry $i_0 \mapsto (x_0, t_0)$.

Case 2: When $\text{lastTerm}(log_v) = \text{lastTerm}(log_1)$ and $\text{length}(log_v) < \text{length}(log_1)$, log_v is a prefix of log_1. Hence, log_v also contains $i_0 \mapsto (x_0, t_0)$.

Therefore, Definition 4 (Leader Completeness) holds under the assumption of Definition 6 (Cluster Well-Formedness).

B. Liveness

Any deterministic consensus protocols including ReCraft, Raft and Paxos cannot guarantee liveness under unrestricted network failures [39], so we show the liveness of ReCraft based on the same assumption as Raft. We assume message delivery to a quorum within a finite amount of retrials and

$$broadcastTime << electionTimeout << MTBF.$$

In this context, $broadcastTime$ refers to the average duration for nodes calling RPCs and receiving responses, while $electionTimeout$ represents the duration between heartbeat messages. $MTBF$ stands for the mean time between failures, which is the average duration between failures for a single node. With these assumptions in place, the liveness of ReCraft is stated in Theorem 2.

Theorem 2 (Liveness): ReCraft responds to all requests from clients in a timely manner.

Proof 2: Proving Theorem 2 requires four case analyses.

1) Termination of entry appends by leaders of each cluster.
2) Termination of the normal leader election process,
3) Progress assurances for subclusters following a split,
4) Termination of a merge when subclusters combine.

Case 1, 2: They are trivial, following the original Raft process and the aforementioned assumptions.

Case 3: This requires additional analysis during the split operation: when the leave split operation is acknowledged and committed in all subclusters, and when it is missed in some subclusters. In the former case, all subclusters needing leader election can immediately start the election. In the latter case, nodes in the missed subclusters will send leader election messages to known peers after the election timeout, which is also a part of the original Raft protocol. This will trigger a pull-based recovery. If all known peers are unavailable, the nodes query the naming service, which we assume always available, to find a node to pull from.

Case 4: Due to the 2PC-style process of our merging operation, as long as the majority of nodes in each subclusters are reachable and alive as in our assumption, proving liveness becomes straightforward.

C. Proof Mechanization

To mitigate the risk of human error, we have utilized Rocq prover [40] to outline and validate our high-level proof structure. This approach provides a stronger guarantee of the proof's correctness. We wrote the mechanized proof structure of ReCraft in Rocq from scratch, but similar to our proof, Raft-related parts follow the structure of existing Rocq proofs that verify Raft [26], [38].

VII. Evaluation

We implement ReCraft in etcd version 3.5 by modifying 4K+ lines of Go code in the etcd-Raft [41] library. We evaluate ReCraft on a public research cloud in a single datacenter region using up to 16 virtual machine (VM) instances, each with 2 vCPUs and 8GB memory, for running etcd and clients. We focus on the write performance, which is our main interest, since the read performance scales easily by adding learners [9]. The block drives of VMs run on Cinder on Ceph, which perform poorly for small writes, so we use 512B requests.

Because etcd does not support split/merge, we compare ReCraft against an emulated split and merge of TiKV/CockroachDB on etcd, which we call TC. We use emulation because both TiKV and CockroachDB only support splitting logical instances of Raft, do not provide control over the physical locations of logical Raft instances, cannot configure Raft instances to have different numbers of nodes, and do not allow three or more clusters split/merge. We realize TC using a script that issues a series of commands from etcd admin tools [42] to go through the same steps as TiKV/CockraochDB split and merge and achieves the same outcome as ReCraft. Note that the split and merge of logical Raft clusters in TiKV/CockroachDB can happen within the same physical nodes. In this case, TiKV/CockroachDB can transfer data more efficiently, but we assume a case where Raft clusters need to be in disjoint physical nodes for reliability and performance.

All experiments are measured at least three times after 30 seconds of warm up time from either the client (Figure 6, Figure 7a, and Figure 8a) or the server driving the reconfiguration (Figure 7b and Figure 8b).

A. Performance Overhead

ReCraft adds new features to etcd, and we evaluate if ReCraft adds any overhead to regular etcd operations. Figure 6 compares the write performance of 3-node ReCraft-etcd and unmodified etcd clusters as we increase the load. Both systems perform identically, showing that ReCraft implementation does not affect the regular workings of etcd.

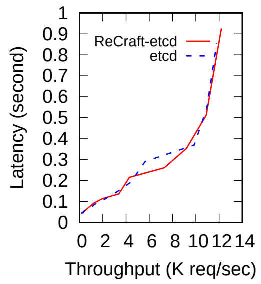


Fig. 6: etcd performance with ReCraft vs Raft.

B. Split Performance

Throughput. Split operations require two consensus steps and do not add noticeable overhead. Figure 7a shows the throughput of 6-node (left) and 9-node (right) ReCraft-etcd instances splitting into two and three 3-node subclusters, respectively. We use 128 clients stressing the cluster with a uniform random puts. There is no performance impact when the split occurs at the 30-second mark. After the split, subclusters divide the load and double and triple the combined throughput.

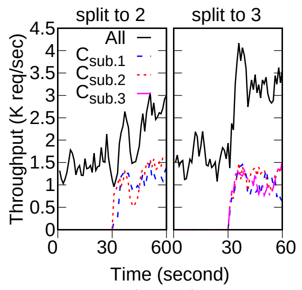
We do not compare the throughput of ReCraft against TC, as the throughput before and after the split/merge for the two systems is identical as they share the same etcd base. The only difference in the throughput measurement is the duration of the throughput dip when the split/merge occurs, which is captured by the latency measurement below.

Latency. We compare the latency for the split operation against TC. TC removes nodes that need to split through a membership change (e.g., 3 nodes are removed to split a 6-node cluster into two subclusters), takes a snapshot of the existing data inside removed nodes, installs snapshot and the subcluster configuration to the nodes, and restarts them as subclusters.

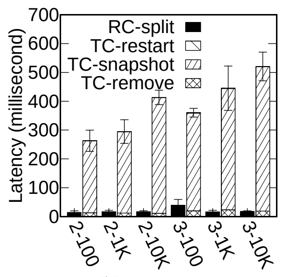
We use the same node configurations as the throughput measurement but with etcd instances holding 100, 1K, and 10K key-value (KV) pairs. ReCraft takes almost constant time to complete the split as it always entails committing two log entries (Figure 7b). However, TC must take multiple steps as described above. TC takes on average 21x longer mainly due to the data migration. Excluding the data migration, the TC runs similarly to ReCraft but TC's split is driven by a single external service which is also a potential single point of failure.

C. Merge Performance

Througput. We merge two and three 3-node clusters into a single 6-node (left) and 9-node (right) clusters, respectively, and measure the throughput (Figure 8a). We use a small amount of load (i.e., 2 clients with uniform random puts) as it is desired to merge clusters when the nodes are underutilized. The merge is blocking so there is a small throughput dip when merging happens at the 30-second mark. Individual clusters before and after the merge perform almost the same due to etcd's extended wait time for batching under a low load: the average latency for each request doubles after the merge.

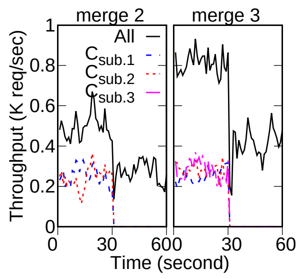


(a) Throughput

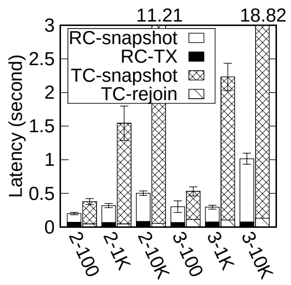


(b) Latency

Fig. 7: Split performance. (a) Throughput of 6-node and 9-node cluster splitting into two (left) and three (right) subclusters, respectively. (b) Split latency for ReCraft (RC) and TiKV/CockroachDB emulation (TC). The x-tic, *a*-*b*, indicates a cluster with *b* KV pairs splitting *a*-ways.



(a) Throughput



(b) Latency

Fig. 8: Merge performance. (a) Throughput of two (left) and three (right) 3-node subclusters merging into one cluster. (b) Merge latency for ReCraft (RC) and TiKV/CockroachDB emulation (TC). The x-tic, *a*-*b*, indicates *a* subclusters with *b* KV pairs merging.

Latency. For a latency evaluation, we compare ReCraft against TC that coalesces all subcluster data in one of subclusters, terminates all subclusters but the one with the coalesced data, and adds all nodes from terminated subclusters to the live one.

We use the same node configurations as the throughput experiment but with etcd instances holding 100, 1K, and 10K KV pairs. The 2PC takes almost constant time for ReCraft across different settings (Figure 8b). For both ReCraft and TC, merging the data takes the longest. Especially, TC exchanges data with more blocking and can take from 1.7x to 20x more time than ReCraft depending on the data size. Similar to the split, ReCraft approach is more robust and performant.

D. Fault Tolerance During Split and Merge

We analyze the fault-tolerance of ReCraft and TC deployments in Sections VII-B and VII-C based on the minimum number of node failures that can completely stop the split and merge (Table I). For TC, we consider a non-replicated cluster

N-way operation	ReCraft Phase 1	ReCraft Phase 2	ReCraft Phase 3	TC CM	TC CM-repl.
Split	$f_{old}+1$	$N(f_{sub}+1)$	-	1	$f_{cm}+1$
Merge	$f_{sub}+1$	$f_{sub}+1$	$f_{sub}+1$	1	$f_{cm}+1$

TABLE I: Minimum number of node failures to completely stop the operation under a uniform subcluster sizes. f is the number of node failures a ReCraft/Raft/etcd cluster can tolerate (old: initial configuration; sub: subcluster configuration; cm: replicated cluster manager configuration).

manager (CM) and a replicated CM (CM-repl) on Raft. The fault tolerance of ReCraft varies by the operation phases, but only cluster granularity interrupts can stop the operation.

For split, C_{old} needs to fail to stop SplitEnterJoint (phase 1), and so all N subclusters have to fail to stop SplitLeaveJoint (phase 2) as subclusters can leave joint mode on their own. Merge needs every subcluster during the 2PC (phases 1 and 2) and the data exchange (phase 3), so disabling one subcluster in any phase is enough to stop the operation. For TC, one can fail the subclusters as in ReCraft, but failing the non-replicated CM or the CM cluster (C_{cm}) is enough. TC with non-replicated CM is the least fault tolerant among all when C_{sub} consists of at least three nodes (i.e., $f_{sub} \geq 1$). If the sizes of C_{sub} and C_{cm} are the same (e.g., 3), ReCraft can tolerate more failures for split and the same number for merge compared to TC. However, recall that ReCraft does not need separate CMs, is more concurrent, and blocks minimally.

E. Membership Change

The performance of the AR-RPC, the JC, and ReCraft membership change is dominated by the number of consensus steps analyzed in Section IV-B. The consensus step to commit a command (e.g., a new configuration) for AR-RPC, JC, and ReCraft takes on average 11.4 ms. ReCraft performs equal to or better than AR-RPC and JC for membership changes under practical cluster sizes between 2 to 5 except for when reducing the cluster size from 5 to 2, which requires one extra consensus step than JC.

VIII. Related Work

Sharding SMR. TiKV [24] and CockroachDB [25] use the multi-Raft design with logical cluster split and merge features (Section II-C). Different from ReCraft, their cluster split and merge operations are driven by the cluster manager, which uses sophisticated distributed locks and transactions and could become a single point of failure. One may choose to replicate the cluster manager for fault tolerance (e.g., as in Section VII-D), but it adds a burden to manage another cluster. ReCraft relies on existing consensus mechanisms of participating clusters which is self-contained, more robust, and simple. Note that ReCraft protocol can be complementarily added to the multi-Raft design for the logical Raft instance management.

Logically partitioning the SMR logs with some overlay logic on a single SMR cluster is another approach for scaling the SMR approach [43]–[46]. ReCraft works at the underlay SMR log and is a complementary design.

Blockchains are SMR logs in Byzantine environments and a few block chain designs employ sharding [47]–[50]. However, their designs and assumptions surrounding reconfigurations are different from those of ReCraft, as blockchains heavily rely on probability and assume fluctuating memberships.

Membership Change Schemes for Consensus-based Distributed Systems. Consensus and similar linearizable distributed systems [12] have been at the center of managing other system configurations and their own reconfiguration has been focused on the single cluster membership change. Most of these reconfiguration protocols [9], [14]–[19], [51], [52] entail blocking until the configuration fully commits or require administrator interventions. Except for the merge operation, ReCraft applies the reconfiguration in a wait-free fashion so other requests can be serviced during the reconfiguration steps.

Membership Change Schemes Based on Atomic Registers. DynaStore [53] and RAMBO [54], [55] propose cluster reconfiguration protocols using distributed atomic registers (e.g., ABD protocol [56]). Atomic registers commit writes atomically using quorums, but they are free from FLP impossibility [39]: while the register is strongly consistent and highly available, the ordering of concurrent writes to it is difficult to control. Thus, membership services use multiple registers that apply partially ordered deltas to existing configurations, and merging the information in the registers yields the final configuration. However, due to the quorum-based nature, these services and ReCraft share commonalities which include the configuration overlap between consecutive configurations (e.g., as in quorum overlap in ReCraft) and the need for consulting multiple configurations (e.g., as in joint quorums in ReCraft) during the reconfiguration process.

Pulling Data. ReCraft's log pulling during the split and merge is inspired by the leader election phase of Paxos [7], [8]. While ReCraft only pulls committed entries for recovering nodes, a closer idea to Paxos of pulling uncommitted log entries is applied to a Raft design in MongoDB [57]. Mirador uses pulling for enhanced modularization and debugging: it leverages pull-based micro-replication to build a Byzantine fault-tolerant SMR protocol [58].

Formal Proofs. Due to their sophisticated nature, Raft and Paxos protocols are formally verified in many studies [17], [28], [38], [59]–[61]. The ReCraft proof relies on the overall safety proof on Raft [38] and Adore's [26] proof of Raft membership changes, which are mechanized using Rocq.

IX. Conclusion

We present ReCraft, a reconfigurable Raft, with a new reconfiguration protocol that splits and merges Raft clusters and efficiently changes the cluster membership. The split and merge protocol is the first self-contained and fault-tolerant design based on the consensus of all relevant subclusters. The ReCraft membership change scheme is more efficient than Raft's approach for practical cluster sizes. We present proof of ReCraft's safety and liveness and demonstrate the efficiency of an etcd-based implementation in a public cloud.

Acknowledgments

We thank the anonymous reviewers for their insightful reviews and José Orlando Pereira for shepherding the paper. We also thank Pruthvi Prakash Navada for testing and packaging the artifact. The artifact (i.e., ReCraft code and Rocq proof sketches) is available at https://zenodo.org/records/15059499. This work is supported by the Khoury Apprenticeship Program, NSF award #2019285, and the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (RS-2025-00556905).

References

[1] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 159–172. [Online]. Available: https://doi.org/10.1145/2043556.2043572

[2] J. Lu, L. Chen, L. Li, and X. Feng, "Understanding node change bugs for distributed systems," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 399–410.

[3] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," in *Proc. of the ACM Symposium on Cloud Computing*, ser. SoCC '14. New York, NY, USA: ACM, 2014, pp. 1–14.

[4] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proc. of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 335–350. [Online]. Available: https://dl.acm.org/doi/10.5555/1298455.1298487

[5] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8, no. 9, 2010.

[6] etcd Authors, "etcd," https://etcd.io/, 2024.

[7] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

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

[9] R. Van Renesse and D. Altinbuken, "Paxos made moderately complex," *ACM Computing Surveys (CSUR)*, vol. 47, no. 3, pp. 1–36, 2015.

[10] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2014, pp. 305–319. [Online]. Available: https://dl.acm.org/doi/10.5555/2643634.2643666

[11] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[12] R. van Renesse, N. Schiper, and F. B. Schneider, "Vive la difference: Paxos vs. Viewstamped Replication vs. Zab," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 4, pp. 472–484, Jul. 2015.

[13] L. Lamport, D. Malkhi, and L. Zhou, "Vertical paxos and primary-backup replication," in *Proceedings of the 28th ACM symposium on Principles of distributed computing*, 2009, pp. 312–313.

[14] ——, "Stoppable paxos," *TechReport, Microsoft Research*, 2008.

[15] ——, "Reconfiguring a state machine," *ACM SIGACT News*, vol. 41, no. 1, pp. 63–73, 2010.

[16] L. Lamport, "Generalized consensus and Paxos," Microsoft, Tech. Rep. MSR-TR-2005-33, Mar. 2005.

[17] J.-Y. Shin, J. Kim, W. Honoré, H. Vanzetto, S. Radhakrishnan, M. Balakrishnan, and Z. Shao, "WormSpace: A modular foundation for simple, verifiable distributed systems," in *Proc. of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: ACM, 2019, pp. 299–311.

[18] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, "The SMART way to migrate replicated stateful services," in *Proc. of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, ser. EuroSys '06. New York, NY, USA: ACM, 2006, pp. 103–115.

[19] B. Liskov and J. Cowling, "Viewstamped replication revisited," MIT, Tech. Rep. MIT-CSAIL-TR-2012-021, Jul. 2012. [Online]. Available: http://hdl.handle.net/1721.1/71763

[20] D. Ongaro, *Consensus: Bridging theory and practice*. Stanford University, 2014.

[21] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machines for WANs," in *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '08. Berkeley, CA, USA: USENIX Association, 2008, pp. 369–384. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855767

[22] M. Whittaker, A. Ailijiang, A. Charapko, M. Demirbas, N. Giridharan, J. M. Hellerstein, H. Howard, I. Stoica, and A. Szekeres, "Scaling replicated state machines with compartmentalization," *Proc. VLDB Endow.*, vol. 14, no. 11, p. 2203–2215, jul 2021. [Online]. Available: https://doi.org/10.14778/3476249.3476273

[23] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proc. of the 24th ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 358–372.

[24] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang *et al.*, "Tidb: a raft-based htap database," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3072–3084, 2020.

[25] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss *et al.*, "Cockroachdb: The resilient geo-distributed sql database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1493–1509.

[26] W. Honoré, J.-Y. Shin, J. Kim, and Z. Shao, "Adore: atomic distributed objects with certified reconfiguration," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 379–394.

[27] Z. Wang, C. Zhao, S. Mu, H. Chen, and J. Li, "On the parallels between Paxos and Raft, and how to port optimizations," in *Proc. of the 2019 ACM Symposium on Principles of Distributed Computing*, ser. PODC '19. New York, NY, USA: ACM, 2019, pp. 445–454.

[28] W. Honoré, J. Kim, J.-Y. Shin, and Z. Shao, "Much ADO about failures: A fault-aware model for compositional verification of strongly consistent distributed systems," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021.

[29] D. Ongaro, "bug in single-server membership changes," https://groups.google.com/g/raft-dev/c/t4xj6dJTP6E/m/d2D9LrWRza8J, 2015.

[30] "Cockroachdb design document," https://github.com/cockroachdb/cockroach/blob/master/docs/design.md#splitting--merging-ranges, 2022.

[31] "Mongodb," https://www.mongodb.com/, 2024.

[32] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 149–160. [Online]. Available: https://doi.org/10.1145/383059.383071

[33] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 161–172. [Online]. Available: https://doi.org/10.1145/383059.383072

[34] N. Azmy, S. Merz, and C. Weidenbach, *A Rigorous Correctness Proof for Pastry*. Springer-Verlag, 2016, pp. 86–101.

[35] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE J.Sel. A. Commun.*, vol. 22, no. 1, p. 41–53, sep 2006. [Online]. Available: https://doi.org/10.1109/JSAC.2003.818784

[36] H. Howard, D. Malkhi, and A. Spiegelman, "Flexible paxos: Quorum intersection revisited," in *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain*, ser. LIPIcs, P. Fatourou, E. Jiménez, and F. Pedone, Eds., vol. 70. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, pp. 25:1–25:14. [Online]. Available: https://doi.org/10.4230/LIPIcs.OPODIS.2016.25

[37] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm (extended version)," https://raft.github.io/raft.pdf, 2024.

[38] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson, "Planning for change in a formal verification of the Raft consensus protocol," in *Proc. of the 5th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP '16. New York, NY, USA: ACM, 2016, pp. 154–165.

[39] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, p. 374–382, Apr. 1985. [Online]. Available: https://doi.org/10.1145/3149.214121

[40] The Rocq development team, "The Rocq prover," https://rocq-prover.org, 2025.

[41] etcd-io, "etcd-raft," https://github.com/etcd-io/raft, 2023.

[42] etcd.io, "etcd Recovery Documentation," https://etcd.io/docs/v3.3/op-guide/recovery/, 2023.

[43] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, "Tango: Distributed data structures over a shared log," in *Proc. of the 24th ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 325–340.

[44] J. Lockerman, J. M. Faleiro, J. Kim, S. Sankaran, D. J. Abadi, J. Aspnes, S. Sen, and M. Balakrishnan, "The fuzzylog: A partially ordered shared log," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 357–372.

[45] M. Balakrishnan, J. Flinn, C. Shen, M. Dharamshi, A. Jafri, X. Shi, S. Ghosh, H. Hassan, A. Sagar, R. Shi, J. Liu, F. Gruszczynski, X. Zhang, H. Hoang, A. Yossef, F. Richard, , and Y. J. Song, "Virtual consensus in Delos," in *Proc. of the 14th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '20, 2020, pp. 617–632.

[46] M. Whittaker, N. Giridharan, A. Szekeres, J. Hellerstein, H. Howard, F. Nawab, and I. Stoica, "[solution] matchmaker paxos: A reconfigurable consensus protocol," *Journal of Systems Research*, vol. 1, no. 1, 2021.

[47] B. David, B. Magri, C. Matt, J. B. Nielsen, and D. Tschudi, "Gearbox: Optimal-size shard committees by leveraging the safety-liveness dichotomy," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 683–696. [Online]. Available: https://doi.org/10.1145/3548606.3559375

[48] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 583–598.

[49] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 931–948. [Online]. Available: https://doi.org/10.1145/3243734.3243853

[50] H. Huang, X. Peng, J. Zhan, S. Zhang, Y. Lin, Z. Zheng, and S. Guo, "Brokerchain: A cross-shard blockchain protocol for account/balance-based state sharding," in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, 2022, pp. 1968–1977.

[51] A. Shraer, B. Reed, D. Malkhi, and F. P. Junqueira, "Dynamic Reconfiguration of Primary/Backup clusters," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 425–437. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/shraer

[52] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with bft-smart," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 355–362.

[53] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer, "Dynamic atomic storage without consensus," *J. ACM*, vol. 58, no. 2, pp. 1–32, Apr. 2011.

[54] N. A. Lynch and A. A. Shvartsman, "RAMBO: A reconfigurable atomic memory service for dynamic networks," in *Proc. of the 16th International Conference on Distributed Computing*, ser. DISC '02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 173–190. [Online]. Available: http://dl.acm.org/citation.cfm?id=645959.676144

[55] S. Gilbert, N. Lynch, and A. Shvartsman, "RAMBO II: rapidly reconfigurable atomic memory for dynamic networks," in *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, 2003, pp. 259–268.

[56] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *Journal of the ACM (JACM)*, vol. 42, no. 1, pp. 124–142, 1995.

[57] S. Zhou and S. Mu, "Fault-Tolerant replication with Pull-Based consensus in MongoDB," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 687–703. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/zhou

[58] T. Distler, M. Eischer, and L. Lawniczak, "Micro replication," in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2023, pp. 123–137.

[59] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, "IronFleet: Proving practical distributed systems correct," in *Proc. of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 1–17.

[60] O. Padon, G. Losa, M. Sagiv, and S. Shoham, "Paxos made EPR: Decidable reasoning about distributed protocols," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 1–31, Oct. 2017.

[61] Á. García-Pérez, A. Gotsman, Y. Meshman, and I. Sergey, "Paxos consensus, deconstructed and abstracted," in *Proc. of the 27th European Symposium on Programming*, ser. ESOP '18, 2018, pp. 912–939.

APPENDIX

Safety proofs

With the approach described in Section VI, three of the listed lemmas - LO (Leader Append-Only), ES (Election Safety), and LM (Log Matching) - are either identical or nearly identical to the corresponding proofs in the original Raft.

Proof 3 (Proofs for LA, ES, and LM): With the epoch number fixed, the proofs closely mirror those of the original Raft algorithm. Each relies on basic quorum-overlap arguments as the core decision procedure. Because both log entry confirmation and leader election decisions are made through quorum agreement, the corresponding safety properties are preserved. The following sections present detailed proofs for each of these properties.

1) **Leader Append-Only**: Once a server becomes leader in a given epoch, it only appends new entries to its log and never overwrites or deletes existing entries. This property holds in ReCraft because the leader exclusively controls log replication for its epoch, and the log update mechanism enforces that all entries are appended to the end of the log. This directly mirrors the append-only guarantee of Raft, where leaders never modify previously committed log entries.
2) **Election Safety**: With the epoch number fixed, elections in ReCraft proceed either through a normal quorum or a joint quorum. Since every joint quorum includes at least one normal quorum as a subset, and since each server votes at most once per epoch, two different leaders cannot be elected in the same epoch. This preserves the core Election Safety invariant of Raft: at most one leader can be elected per term. The argument follows the same structure as in Raft—vote uniqueness, quorum intersection, and monotonic term progression ensure safety even with the generalized quorum mechanism.
3) **Log Matching**: With a fixed epoch, ReCraft preserves the Log Matching Property: if two logs contain an entry with the same index and term, then the logs are identical in all preceding entries. The proof closely follows the original Raft argument. Log replication in ReCraft ensures that followers only accept entries that match their existing logs at the given index and term. This prevents divergence and guarantees that logs with a common suffix are consistent in their prefixes as well. As in Raft, this property plays a key role in ensuring state machine consistency.

However, leader completeness (Definition 4) requires significant alteration compared to the original Raft due to the modifications introduced by ReCraft, and the proof is described in Section VI. Within the proof, we use Definition 6 as part of it. The main purpose of Definition 6 is showing that all the cooperating nodes of that epoch share the cluster configuration. For example, consider a situation where a follower a of epoch e and cluster configuration C_a accepted a message from a leader p with cluster configuration C_p.s The following entails detailed proofs related to Definition 6 (Cluster Well-Formedness).

Proof 4: The proof is conducted through strong induction on the epoch number. Suppose there are nodes a_1 and a_2 with the same epoch number e, where their assigned clusters are C_1 and C_2, respectively. In this case, $C_1 = C_2 \lor C_1 \# C_2$ when $\#$ stands for "disjoint" set.

- If $C_1 \# C_2$, then the goal immediately holds.
- If $C_1 \# C_2$, let's select a node a that satisfies $a \in C_1 \cap C_2$. Find a chain of reconfigurations $C_{10} \to C_{11} \to \cdots \to C_{1m} = C_1$ and $C_{20} \to C_{21} \to \cdots \to C_{2n} = C_2$ where C_{10} and C_{20} are initial clusters, and a is a member of all the constituting clusters of the two chains. With the fact of the initial condition, $C_{10} = C_{20} \lor C_{10} \# C_{20}$ holds, but the second case is a contradiction since $a \in C_{10}$ and $a \in C_{20}$. Thus $C_{10} = C_{20}$. By using Definition 7 with induction hypothesis (Cluster Well-Formedness) on epoch 0, we know that Definition 7 holds on epoch 0. From this, there is only one possible reconfiguration for $C_{10} = C_{20}$. Since $C_{10} \to C_{11}$ and $C_{20} \to C_{21}$, C_{11} and C_{21} are resulting clusters of the same reconfiguration. But then $C_{11} = C_{21}$, since resulting clusters of a reconfiguration must be disjoint each other, and $C_{11} \# C_{21}$ because of a. Also, the induction hypothesis on epoch 1 can derive the fact, $C_{12} = C_{22}$, via a similar process. Repeating this process, we can show that $m = n$ and $C_{1m} = C_{2n}$. Thus $C_1 = C_2$.

Therefore, Definition 6 holds.

With all ingredients discussed, we finally prove Definition 7 (Log Consistency).

Proof 5: Let $i \mapsto (x_1, t_1)$ and $i \mapsto (x_2, t_2)$ be the two committed entries. Then there must be directly committed entries $i'_1 \mapsto (x'_1, t'_1)$ and $i'_2 \mapsto (x'_2, t'_2)$ that lead to the commit of each $i \mapsto (x_1, t_1)$ and $i \mapsto (x_2, t_2)$. Utilize Definition 3 and Definition 4 with directly committed entries, and we can conclude that $(x_1, t_1) = (x_2, t_2)$.

Now we show that all sub-definitions that are necessary to show our main safety statement, but most of them still rely on Definition 6. Therefore, we show Definition 6. Indeed, the proof structure is mutually recursive; Definition 7 of epoch e depends on Definition 6 of epoch e, and Definition 6 of epoch e depends on Definition 7 of all the previous epochs.

By combining Definitions 6 and 7, we can prove that Definition 7 holds on every epoch. The top level theorem, state machine safety (Theorem 1), is a corollary of this fact.

A. *Proof mechanization*

As mentioned in Section VI-C, we have structured the proof using Rocq to validate our high-level proof structure and mitigate the risk of human errors in this and other complex parts of the system. We have intentionally omitted several trivial or tedious subproofs, such as the monotonicity of increasing term and epoch numbers for a single node.

Using Coq for this purpose presents both advantages and challenges. On one hand, mechanization provides a high degree of confidence in the correctness of our reasoning. On the other hand, it introduces additional complexity to the proof process, as it requires the development of multiple formal definitions that precisely capture the system state and are expressed using Coq definitions with several auxiliary annotations.

Our long-term goal is to achieve a higher level of assurance than the current version. In particular, we aim to complete all the tedious and detailed invariant proofs that are currently marked with `admit` in our Coq development. Completing these will yield a fully verified proof with no remaining gaps.