Omni-Paxos: Breaking the Barriers of Partial Connectivity

Harald Ng
hng@kth.se
KTH Royal Institute of Technology

Seif Haridi
haridi@kth.se
KTH Royal Institute of Technology
RISE Research Institutes of Sweden

Paris Carbone
parisc@kth.se
KTH Royal Institute of Technology
RISE Research Institutes of Sweden

Abstract

Omni-Paxos is a system for state machine replication that is completely resilient to partial network partitions, a major source of service disruptions in recent years. Omni-Paxos achieves its resilience through a decoupled design that separates the execution and state of leader election from log replication. The leader election builds on the concept of quorum-connected servers, with the sole focus on connectivity. Additionally, by decoupling reconfiguration from log replication, Omni-Paxos provides flexible and parallel log migration that improves the performance and robustness of reconfiguration. Our evaluation showcases two benefits over state-of-the-art protocols: (1) guaranteed recovery in at most four election timeouts under extreme partial network partitions, and (2) up to 8x shorter reconfiguration periods with 46% less I/O at the leader.

CCS Concepts: • **Computer systems organization** → **Availability**; *Distributed architectures*; Reliability.

Keywords: consensus, state machine replication, partial connectivity, reconfiguration

ACM Reference Format:
Harald Ng, Seif Haridi, and Paris Carbone. 2023. Omni-Paxos: Breaking the Barriers of Partial Connectivity. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3552326.3587441

1 Introduction

In recent years, Replicated State Machines (RSMs) have become crucial in building reliable stateful services at scale. Examples include cluster management [12, 38], distributed coordination [6, 23], high-availability in data management [9–11], data replication [20, 36, 41], and lock services [18], among others. Despite their critical role, recent outages reveal fundamental flaws in the core design of such protocols when special network conditions arise. In particular, recent analyses [24, 30] on Cloudflare's outage brought attention to the problem of RSMs and partial connectivity. i.e., when network partitions occur only partially at the link level between servers. We argue that such failure conditions serve as a good foundation to question the resilience of existing RSMs beyond conventional failure models and inspire the creation of protocols with improved resilience and performance.

Omni-Paxos is an RSM system that supports leader election, log replication, and reconfiguration while guaranteeing stable progress with partial connectivity. Omni-Paxos only requires a single server connected to a majority to make progress while existing protocols impose the need for a fully-connected majority. The increased resilience of Omni-Paxos is attributed to its modular design, which allows each of the core components of the system (leader election, log replication, reconfiguration) to be designed and optimized in isolation. This is in contrast to existing RSMs that follow a monolithic single-protocol design. Omni-Paxos' leader election component integrates connectivity to elect a quorum-connected server i.e., a server that can form a majority quorum for log replication. It automatically detects and switches to a capable leader when required to cope with changes in network connectivity. To guarantee progress with partial connectivity, the log replication protocol complements this with a synchronization phase such that even a trailing but quorum-connected server can become the leader and maintain a consistent log. This increases Omni-Paxos resilience to only require a single quorum-connected server to make progress, rather than a fully-connected quorum. Moreover, reconfiguration decouples log migration from log replication. This lets other servers than the leader migrate the log to new servers, improving performance and robustness of reconfiguration.

The concept of decoupling logic in an RSM has been previously explored in various works [25, 35, 37, 39, 40], yet, none of these ideas provide a complete RSM specification but rather focus on optimizing a single component (e.g., log replication or reconfiguration). In contrast, prior attempts to provide a complete specification [31, 34] have resulted

EuroSys '23, May 8–12, 2023, Rome, Italy

ACM ISBN 978-1-4503-9487-1/23/05.
https://doi.org/10.1145/3552326.3587441

Table 1. Comparison of protocols and partial connectivity (*Addressed in recent PV/QC [24] Raft patch). Omni-Paxos guarantees progress with one QC server, while other protocols require at least a majority to overcome all the partial connectivity scenarios.

	Protocol Properties (QC= quorum-connected, EQC=elected by quorum-connected)					**Partial-Connectivity Scenarios** (✓:stable progress, ✗:unavailable)		
	Log Sync Phase	Candidate Requirements	Leader Vote Gossiping	QC Status Heartbeats	Guaranteed Progress Requirement (#QC servers)	Quorum-Loss Scenario	Constrained Election Scenario	Chained Scenario
Multi-Paxos [28, 37]	✓	QC	✓		$\geq \lceil N/2 \rceil$	✗	✓	✗
Raft [34]		QC + max log	✓		$\geq \lceil N/2 \rceil$	✗*	✗	✗*
VR [31]	✓	QC + EQC	✓		$\geq \lceil N/2 \rceil$	✗	✗	✗
Zab [25, 33]		QC + EQC + max log	✓		$\geq \lceil N/2 \rceil$	✗	✗	✗
Omni-Paxos	✓	QC		✓	≥ 1	✓	✓	✓

in monolithic designs. Thus, protocols either omit cross-component optimization or suffer from problems due to inter-dependency between different logical components (as shown in §2). Omni-Paxos bridges this gap by providing a complete RSM system that benefits from a decoupled design to improve both resilience and performance. Omni-Paxos aligns with recent trends in highly decoupled systems that build on RSM services. For example, Corfu [32] decouples storage from compute while Delos [17] and Scalog [21] decouple reconfiguration from log replication using log virtualization.

Contributions. In this paper, we claim the following:

- We demonstrate liveness issues in existing RSM protocols under partial connectivity and discuss the causes behind them.
- We define Quorum-Connected Leader Election, a specification that guarantees progress in Paxos-based protocols even with partial connectivity.
- We present Omni-Paxos, an RSM system that provides complete resilience against partial connectivity and flexible reconfiguration. We provide a full description of the system along with the specification and correctness arguments.
- We present an experimental evaluation that includes the following highlights compared to state-of-the-art protocols: (1) Constant-time recovery from any partial network partition. Omni-Paxos recovers in constant time in scenarios where other protocols livelock or deadlock, and (2) Parallel log migration in reconfiguration to provide 8x shorter reconfiguration periods and 46% less I/O at the leader.

2 The Case of Partial Connectivity

Failures in modern networks can be more complex and unforeseen than the ones commonly assumed in systems research literature. Partial connectivity is such a type of failure, where two servers are disconnected while both are still reachable by a third server. An increasing set of studies [15, 16] attribute partial network partitions to network upgrades [3], firewall or network misconfigurations [2], and flaky links between switches [1].

The resilience of RSMs relates to how leader election interacts with log replication to make progress during potential network partitions. To understand how partial connectivity challenges resilience, we first define the common and differentiating properties that exist in the leader election of the most widely-used protocols.

Leader Election in Log Replication: Leader election is required to guarantee progress in log replication protocols [19], and it can be performed using explicit messages to request and submit votes, or implicitly as the initial phase of the log replication protocol itself. For instance, protocols such as Raft [34] and VR [31] use explicit voting (with RequestVote and DoViewChange messages), while a Multi-Paxos [28] server establishes its "leader status" by collecting a majority of promises (P1B messages). Regardless of the approach, servers only vote for a leader with a higher term (also known as ballot or view number), and the leader can only proceed and subsequently commit new entries if it manages to collect a majority of votes. Thus, there is no distinction between electing a leader through an explicit election protocol and collecting a majority of promises in the initial phase of the replication protocol. Instead, we can generalize the requirement for obtaining leadership: A server can only be elected as the leader if it is *quorum-connected* (QC), i.e., directly connected to a majority of servers (including itself). Apart from QC, different protocols may make additional requirements in leader election.

In Raft [34], the elected leader must additionally have the maximum log, i.e., the log with the highest term number among a majority. This allows Raft to remain correct without using an initial synchronization phase where followers transfer missing log entries to a newly-elected leader. In VR [31], a server can only vote for a candidate (by sending DoViewChange) if itself is QC. Thus, a server must not

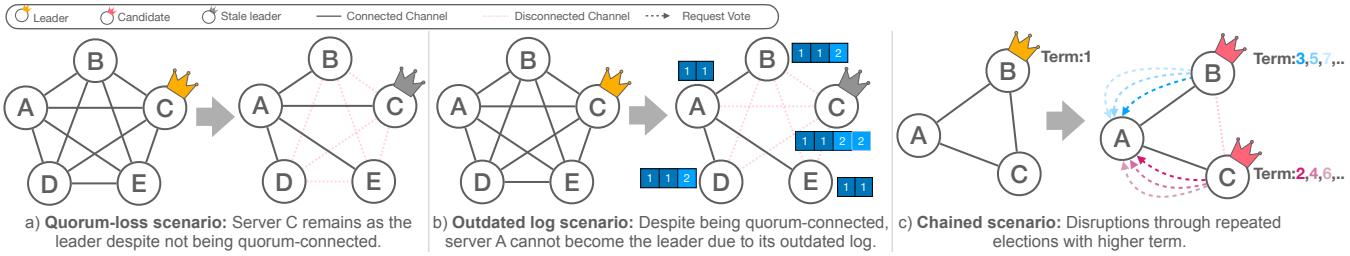


Figure 1. Deadlock and livelock in replicated state machines due to partial connectivity.

only be quorum-connected, but it must also be elected by quorum-connected servers (EQC) to become the leader. That is, a leader must receive votes from a majority of QC servers. Zab [25, 33] adopts both the max log and EQC requirements. Multi-Paxos [28] does not have any additional requirements than QC. Multi-Paxos typically uses failure detectors where followers that suspect leader failure will increment their ballot number and try to take over leadership [37]. A summary of the properties of the mentioned protocols is shown in Table 1.

Partial Connectivity and Leader Election: We now identify three scenarios of partial connectivity that can obstruct RSMs from making progress. The last scenario has been documented in practice causing 6+ hours of down-time for Cloudflare in 2020 [24, 30].

a) Quorum-Loss Scenario. Assume a cluster of five servers that is initially fully-connected and server C is the elected leader. A partial network partition could cause a scenario where all servers are connected to server A, but disconnected from the rest, as depicted in Figure 1a. That makes A the only server that is quorum-connected and therefore the only qualified leader candidate. However, since A is still connected to its leader C, it will not initiate a new election to become the leader. On the other hand, the other servers will attempt a leader change but not get enough votes as they are not QC. As a result, no leader will be elected and log replication progress will suspend. Raft is an exception to this case, server A will learn higher term numbers from the disconnected followers and eventually be elected. However, the randomized timers in Raft might cause other servers to continuously disrupt with higher terms and cause unavailability before A is elected. In general, this scenario shows that the alive status of the current leader is an insufficient metric to solely trigger a leader change. Quorum-loss could result in having a leader that is alive but incapable of making progress.

b) Constrained Election Scenario. Consider now the same scenario but with leader C completely partitioned from the rest. Figure 1b illustrates such a scenario including the log of each server. Each log entry carries the term number it was replicated with. Again, we notice that A is the only quorum-connected server. Contrary to the previous scenario, A now observes that C is unreachable and attempts a leader election. Despite having the capability to be elected and make progress, A would not get elected in state-of-the-art protocols such as Raft, Zab, and VR. This is due to the constraints that these protocols impose on candidates in addition to quorum-connectivity. For example, having the max log is an extra requirement in Raft and Zab. In the example, A's last entry has a lower term than B and D, and it will therefore not get voted by them. However, B and D do not qualify either since they are not quorum-connected. VR and Zab further constrain the set of possible candidates to servers that can be elected by other quorum-connected servers (EQC). In this case, no other server apart from A is quorum-connected which implies that no server can be EQC. Both of these cases lead to progress violations where a new leader cannot be elected due to the constraints added on top of quorum-connectivity in leader election.

c) Chained scenario. Figure 1c illustrates a scenario where a link has broken down in a cluster of 3 servers, such that the servers are connected in a chain. Server B is the leader before the link between B and C is disconnected. As C does not receive any messages from B, it suspects that B has failed and increments its term number (to become the leader). If B then observes via A that the leadership has changed, the described scenario will re-occur in the reversed direction; B will suspect that C has failed and become the leader with a higher term. As such, a chained scenario will cause a livelock where the leader repeatedly changes due to a higher term number getting gossiped. This form of gossiping the current leader term occurs at different stages in different protocols. In Multi-Paxos and Raft, server A would first elect the server with a higher term, and when the old server tries to replicate entries, A will reject it and reply with the new term number. In Zab, once a server elects a new leader, this information is forwarded to all its peers. Whereas in VR, a server that suspects the leader has failed or gets notified about it, will propose a leader change that in turn gets forwarded by all other peers. Chained scenarios could be resolved in some of the existing protocols if a fully-connected server (e.g., A in this case) manages to get elected, e.g., due to some additional constraints such as the log progress in Raft or pre-determined leader ordering in VR. However, in scenarios where there

is no fully-connected server (e.g., a chained scenario with 5 servers), the cluster will be in a livelock with repeated leader changes due to the terms being gossiped.

Key Observations: The described scenarios epitomize the challenges of partial connectivity for RSMs. The quorum-loss scenario shows that leader election protocols equivalent to failure detectors are not sufficient. Partial connectivity could result in situations where the leader is alive but not quorum-connected anymore and thus unable to make progress. The constrained election scenario shows that leader election must not have any strict requirements on servers apart from quorum-connectivity. A server should be an eligible candidate as long as it is connected to a majority. An RSM protocol must therefore complement this with a synchronization phase since a newly-elected leader might not have all the committed entries. Furthermore, the chained scenario shows that gossiping the identity of the current leader could cause liveness issues due to repeated elections originating in servers having inconsistent views on which servers are alive. The properties and capability with partial connectivity of the described protocols are summarized in Table 1.

To modify the existing protocols to handle partial connectivity is not trivial due to their monolithic designs. The correctness of Raft is directly related to the max log property. Raft has no synchronization phase since the leader must have the most updated log. VR has a monolithic design where the view change and reconfiguration logic is tightly coupled with log replication and based on the assumption that a majority have the same view on who should be the leader. The tight coupling between log replication, leader election, and reconfiguration in these protocols implies that it is not possible to modify the logic of one of them independently.

In the rest of this paper, we present Omni-Paxos, an RSM system with a clear separation of concerns as the main design principle used to achieve resilience and stable performance. As detailed in Table 1, Omni-Paxos is resilient to all types of partial connectivity and provides progress even when only a single QC server exists in the cluster, compared to other protocols that require at least a majority in all or specific scenarios. This is achieved by omitting preconditions and gossiping information that are irrelevant, which follows Omni-Paxos' design principle that distills the essential requirements of leader election, log replication, and reconfiguration into separate components. Log replication is only responsible for maintaining a consistent log. Leader election focuses on electing a leader with adequate connectivity to make progress in log replication. Reconfiguration provides fast and efficient migration to new servers while preserving safety across configurations.

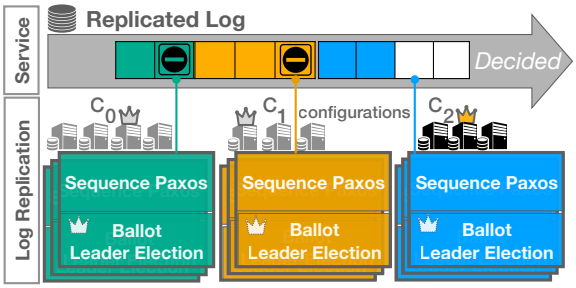


Figure 2. Architecture of Omni-Paxos.

3 System Overview

Omni-Paxos is a system that provides RSM functionalities via three context-independent components with clear objectives, namely log replication (§4), leader election (§5) and reconfiguration (§6). As depicted in Figure 2, Omni-Paxos provides the view of a single consistent replicated log which is accessible through a *service layer*, where every server in Omni-Paxos stores its local copy of the replicated log. The replicated log is populated by a single *Sequence Paxos* instance at a time. Sequence Paxos is a leader-based protocol that maintains the correctness and consistency of the replicated log. A Sequence Paxos instance is accompanied by its own *Ballot Leader Election* (BLE). BLE is responsible for electing a leader and providing stable progress in Sequence Paxos even when only a single QC server exists due to extreme partial connectivity. Upon reconfiguration, the current Sequence Paxos instance is first stopped before the new one takes over. The service layer is responsible for transitioning to the new configuration safely. This includes migrating the log to new servers and starting the new Sequence Paxos and BLE instances in the new configuration.

Preliminaries. We assume the fail-recovery model where servers might fail (non-byzantine) and eventually recover after some arbitrary time. A correct server is a server that might fail and recover a finite number of times. State stored in non-volatile storage is recoverable. We assume a partially synchronous model where messages can be dropped and delayed, but there are long enough periods of synchrony for algorithms to make progress. Servers use bidirectional links to exchange messages. For simplicity, we assume session-based FIFO perfect links. In practice, we use TCP (session drops are handled in §4.1.3). Lastly, partitions could cause a set of links to be temporarily down as discussed in §2. During this period, messages are systematically dropped.

4 Sequence Paxos - Log Replication

In this section, we present Sequence Paxos, the log replication protocol in Omni-Paxos. To guide the design of Sequence Paxos, we present the *Sequence Consensus* properties that are inspired by Generalized Consensus [26]:

SC1. Validity: If a server decides on a log L then L only contains proposed commands.
SC2. Uniform Agreement: For any two servers that decided logs L and L' respectively then one is the prefix of the other.
SC3. Integrity: If a server decides on a log L and later decides on L' then L is a strict prefix of L'.

As seen from the properties, Sequence Paxos is a protocol that replicates a log in strict sequential order without gaps. As argued by Raft [34], this approach leads to a more understandable protocol which has also proven to be practical with high adoption in the industry [22, 23, 36, 41].

4.1 Protocol Overview

In Sequence Paxos, log replication is performed in rounds that are monotonically increasing and uniquely identified by ballot numbers. Servers include their current ballot number when exchanging messages which allow for detecting obsolete messages caused by servers transitioning to different rounds asynchronously. A server can act as a *follower* or the *leader*. The leader of a round is elected through Ballot Leader Election (§5.2), which is performed concurrently and in isolation from the log replication. If the leader detects a higher round, it reverts back to being a follower. The leader handles client requests and appends them to its log before replicating them to the followers. An entry that is appended to the local log of a server is called *accepted*. A *chosen* entry is accepted by a majority and guaranteed to eventually be *decided*. A decided entry cannot be reverted. As the entries are replicated in FIFO order, deciding an entry also decides any preceding entries in the log.

A round in Sequence Paxos consists of two phases: *Prepare* and *Accept*. The Prepare phase consists of a log synchronization such that no chosen entries from previous rounds are lost. Upon completion of the Prepare phase, a majority of servers have synchronized logs and promised to not accept entries from lower rounds. Stragglers outside the majority will synchronize later when they promise. In the Accept phase, the leader replicates entries in FIFO order to the promised followers.

4.1.1 Prepare Phase - Log Synchronization.

Since leader election is independent of log replication, a newly-elected leader could be lagging behind in the log. Thus, it must first catch up with all chosen entries by adopting the most updated log among a majority. Servers can determine each other's missing entries by inspecting the other's variables `acceptedRnd`, `decidedIdx` and log length denoted by `logIdx` (Figure 3b ①). The server with the highest `acceptedRnd` has the most updated log. If two servers have the same `acceptedRnd`, then the longest log is more updated. By exchanging these variables, the servers can determine who is more updated and synchronize their logs by sending the entries that the other is missing.

As seen in Figure 3a, the leader of a new round initiates the log synchronization by sending ⟨Prepare⟩ with those variables to all followers ②. A follower replies with its corresponding variables and the leader's missing log entries in a ⟨Promise⟩ message ③. By promising, the follower will not accept anything from a lower round. Upon receiving a majority of promises, the leader adopts the most updated log (e.g., $[1, 2, 3]$ in Figure 3a). This log is guaranteed to contain all chosen entries and gets synchronized to every promised follower using ⟨AcceptSync⟩ ④. An important invariant is that after a follower has accepted the ⟨AcceptSync⟩ ⑤, its log is guaranteed to be a prefix of the leader's log.

4.1.2 Accept Phase - Log Replication.

After synchronizing the logs in the Prepare phase, the Accept phase becomes straightforward. As seen in ⑥ - ⑨ of Figure 3, the leader appends new client requests to its log and replicates them to the promised followers. When a majority has accepted a certain index in the log, it is decided. Using the FIFO link property, the leader can pipeline entries to the followers without waiting for preceding log entries to be decided.

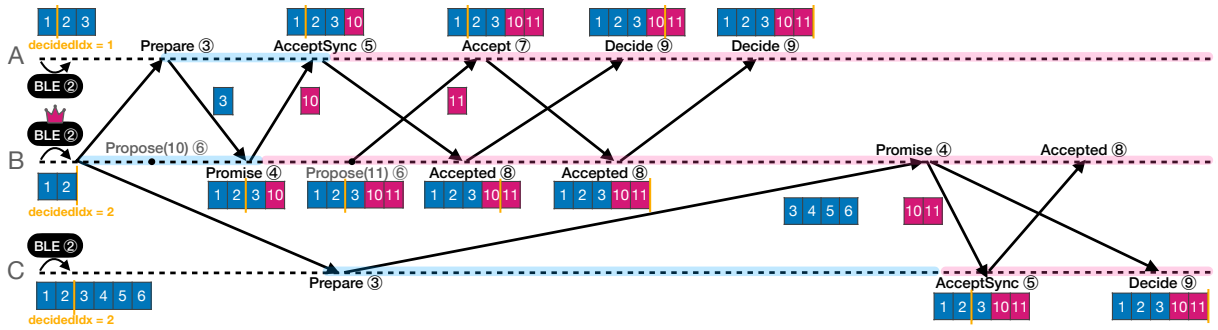
In the Accept phase, the leader also needs to handle promises from followers that were not part of the majority in the Prepare phase (e.g., server C in Figure 3a). In that case, the leader sends ⟨AcceptSync⟩ to synchronize the follower with its current log. Non-chosen entries in that follower's log could be overwritten (e.g., $[4, 5, 6]$ in Figure 3a).

4.1.3 Failure-Recovery and Link Session Drop.

When a server recovers from a failure, it must first perform log synchronization to obtain any entries that might have been decided during the failure. Thus, it loads the persistent variables and goes into *recover* state where it can only handle leader events or ⟨Prepare⟩ messages, both leading to a subsequent log synchronization. During the failure, another leader might have been elected and already completed the Prepare phase. Since the recovering server is unaware of who is the current leader, it sends ⟨PrepareReq⟩ to all its peers ⑩. A receiving server that is the leader responds with ⟨Prepare⟩ ⑪. From this point, the protocol proceeds as usual. Link session drops between two servers are handled similarly. Since they might be unaware that the other has become the leader during the disconnected period, a ⟨PrepareReq⟩ is sent ⑫.

4.2 Correctness and Relation to Paxos

Sequence Paxos shares the terminology and core design of single-value (decree) Paxos [28]. Essentially, the Sequence Consensus properties (§4) are modified from the safety properties of single-value consensus to be a better fit for a replicated log. In both single-value consensus and Sequence Consensus, only proposed values can be decided (SC1). However, rather than agreeing on a single value, Sequence Consensus agrees on a single growing sequence of values (SC2 and



(a) Example execution with 3 servers. The blue and red areas mark the Prepare and Accept phase respectively.

① State and Functions

Persistent state on all servers:

log[]	log with entries (0-indexed)
promisedRnd	the round a server has promised to not accept entries from any lower round
acceptedRnd	the latest round a server has accepted entries in
decidedIdx	the log index that a server has decided up to

Volatile state on all servers:

state	the role and phase a server is in. Initially (FOLLOWER, PREPARE)

Volatile state of leader:

currentRnd	the round that this server is leading in
promises{}	set of received promises
maxProm	the highest promise received during the prepare phase
accepted[]	the accepted index per server. Initialized to 0 for all servers
buffer[]	buffer for client requests received during the prepare phase

Functions:

`stopped()`	true if last entry in the log is SS else false (§5)
`prefix(idx)`	the entries in the log with index {0..idx-1}
`suffix(idx)`	if idx > |log| then [] else the entries in the log with index {idx..|log|-1}

② ⟨Leader⟩ from BLE

Fields:

s	the elected server
n	the round s got elected in

Leader implementation (if *s* = self AND *n* > promisedRnd):

1. reset all volatile state of leader
2. state ← (LEADER, PREPARE), currentRnd ← *n*, promisedRnd ← *n*
3. insert own promise: (acceptedRnd, |log|, self, decidedIdx, `suffix`(decidedIdx)) to promises
4. send⟨Prepare, currentRnd, acceptedRnd, |log|, decidedIdx⟩ to all peers

⑩ Upon Recovery

Receiver implementation:

1. reload log, promiseRnd, acceptedRnd and decidedIdx from persistent storage
2. state ← (FOLLOWER, RECOVER), send ⟨PrepareReq⟩ to all peers

⑪ ⟨PrepareReq⟩ from follower *f*

Receiver implementation:

1. return if state ≠ (LEADER, _)
2. send ⟨Prepare, currentRnd, acceptedRnd, |log|, decidedIdx⟩ to *f*

⑫ ⟨Reconnected⟩ to server *s*

Receiver implementation:

1. if *s* is the current leader then state ← (FOLLOWER, RECOVER)
2. send ⟨PrepareReq⟩ to *s*

④ ⟨Promise⟩ from follower *f*

Fields:

n	promised round
accRnd	the acceptedRnd of *f*
logIdx	the log length of *f*
decIdx	the decidedIdx of *f*
sfx	suffix of entries the leader might be missing

Receiver implementation:

1. return if *n* ≠ currentRnd
2. insert (*accRnd*, *logIdx*, *f*, *decIdx*, *sfx*) to promises

If state = (LEADER, PREPARE) then:

- **P1.** return if |promises| < majority
- **P2.** maxProm ← the value with highest accRnd in promises (and highest logIdx if equal)
- **P3.** if maxProm.accRnd ≠ acceptedRnd then log ← `prefix`(decidedIdx)
- **P4.** append maxProm.sfx to the log
- **P5.** if `stopped()` then clear buffer else append buffer to the log
- **P6.** acceptedRnd ← currentRnd, accepted[self] ← |log|, state ← (LEADER, ACCEPT)
- **P7.** foreach *p* in promises: let *syncIdx* ← if *p*.accRnd = maxProm.accRnd then *p*.logIdx else *p*.decIdx, send ⟨AcceptSync, currentRnd, `suffix`(syncIdx), syncIdx⟩ to *p.f*

If state = (LEADER, ACCEPT) then:

- **A1.** let *syncIdx* ← if *accRnd* = maxProm.accRnd then maxProm.logIdx else decIdx
- **A2.** send ⟨AcceptSync, currentRnd, `suffix`(syncIdx),
- **A3.** if *decidedIdx* > decIdx then send ⟨Decide, currentRnd, *idx*⟩ to *f*

⑥ Proposal *C* from client

Receiver implementation:

1. return if `stopped()`

If state = (LEADER, PREPARE) then:

- **P1.** insert *C* into buffer

If state = (LEADER, ACCEPT) then:

- **A1.** append *C* to the log, set accepted[self] ← |log|
- **A2.** send ⟨Accept, currentRnd, *C*⟩ to all promised followers

⑧ ⟨Accepted⟩ from follower *f*

Fields:

n	promised round
logIdx	the position in the log *f* has accepted up to

Receiver implementation:

1. return if currentRnd ≠ *n* OR state ≠ (LEADER, ACCEPT)
2. accepted[*f*] ← *logIdx*
3. if *logIdx* > decidedIdx AND a majority has accepted *logIdx* then decidedIdx ← *logIdx*, send ⟨Decide, currentRnd, decidedIdx⟩ to all promised followers

③ ⟨Prepare⟩ from leader *l*

Fields:

n	round of leader *l*
accRnd	the acceptedRnd of *l*
logIdx	the length of the leader's log
decIdx	the decidedIdx of *l*

Receiver implementation:

1. return if promisedRnd > *n*
2. state ← (FOLLOWER, PREPARE)
3. promisedRnd ← *n*
4. let *sfx* ← if acceptedRnd > *accRnd* then `suffix`(*decIdx*) else if acceptedRnd = *accRnd* then `suffix`(*logIdx*) else []
5. send ⟨Promise, *n*, acceptedRnd, |log|, decidedIdx, *sfx*⟩ to *l*

⑤ ⟨AcceptSync⟩ from leader *l*

Fields:

n	round of leader *l*
sfx	entries to be appended to the
syncIdx	the position in the log where *sfx* should be appended at

Receiver implementation:

1. return if promisedRnd ≠ *n* OR state ≠ (FOLLOWER, PREPARE)
2. acceptedRnd ← *n*, state ← (FOLLOWER, ACCEPT)
3. log ← `prefix`(*syncIdx*), append *sfx* to the log
4. send ⟨Accepted, *n*, |log|⟩ to *l*

⑦ ⟨Accept⟩ from leader *l*

Fields:

n	round of leader *l*
C	client request

Receiver implementation:

1. return if promisedRnd ≠ *n* OR state ≠ (FOLLOWER, ACCEPT)
2. append C to the log, send ⟨Accepted, *n*, |log|⟩ to *l*

⑨ ⟨Decide⟩ from leader *l*

Fields:

n	round of leader *l*
decIdx	position in the log that has been decided

Receiver implementation:

1. if promisedRnd = *n* AND state = (FOLLOWER, ACCEPT) then decidedIdx ← *decIdx*

(b) State and message handlers.

Figure 3. Log Replication in Omni-Paxos.

SC3). As a result, we can modify and use the invariants of Paxos [28] to prove the correctness of Sequence Paxos. We also provide a formal proof in Appendix A and a PlusCal/TLA+ specification online[1].

The Paxos P1 and P1a invariants [28] remain unchanged: a server can only accept a proposal if it has not promised in a higher round. Invariant P2 guarantees that only a single value can be decided in Paxos: *If a proposal with value v is chosen, then every higher-numbered proposal that is chosen has the value v.* When adapted for a log, the corresponding invariant should guarantee that the log remains consistent and chosen entries must not be removed at a later time. P2 for Sequence Paxos is thus modified to: *If a proposal with sequence v is chosen, then every higher-numbered proposal that is chosen has v as a prefix.* To guarantee this, Sequence Paxos maintains P2c, which states that any proposal issued must be a sequence that extends the highest-numbered sequence accepted among a majority of servers. Sequence Paxos follows the design of Paxos and guarantees P2c by performing the Prepare phase. Instead of adopting the chosen value, Sequence Paxos adopts the chosen sequence, that is, the log with all chosen entries. This is guaranteed to be in at least one server among any majority. The leader then synchronizes this log with all promised followers using ⟨AcceptSync⟩ as described in §4.1.1. New client requests are appended to this log and replicated to the promised followers in FIFO order in the Accept phase. Thus the replicated log will be an extension to the highest-numbered log that was found among a majority in the Prepare phase and P2c will be preserved. □
Note that the Sequence Paxos invariants are described using sequences (i.e., entire log). By using session-based FIFO perfect links and log synchronization, the correctness is preserved even by only sending suffixes.

5 Leader Election

5.1 Quorum-Connected Leader Election

Leader election is crucial for ensuring progress in log replication. In general, leader election protocols elect some correct server, assuming that servers are fully connected within a partition. However, the possibility of partial connectivity imposes a restriction on the set of correct servers that can make progress in log replication. We therefore introduce the concept of *quorum-connected* (QC) servers to derive a set of properties that elects a leader which is not just correct, but also able to form a majority quorum. More formally, a quorum-connected server is *a server that is correct and has a direct link to at least a majority of correct servers (including itself).* The properties of leader election can now be defined as follows:

LE1. QC-Completeness: Eventually, every quorum-connected server elects some quorum-connected server, if a quorum-connected server exists.

[1] https://github.com/haraldng/omnipaxos-tla

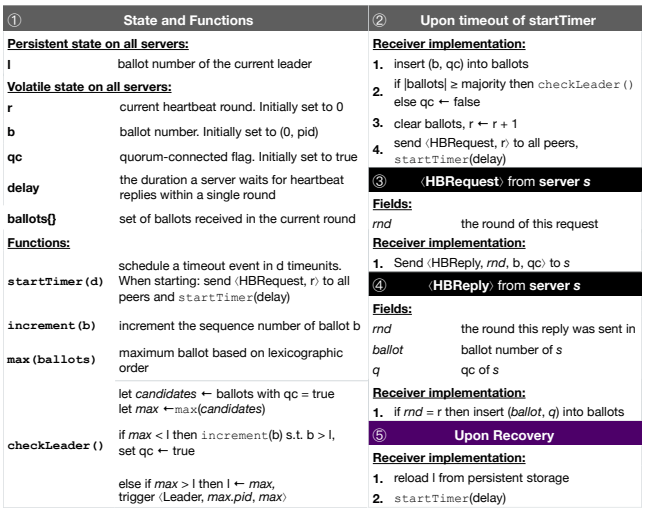


Figure 4. Ballot Leader Election Algorithm.

LE2. QC-Eventual Agreement: Eventually, there is a majority of servers S where no two quorum-connected servers in S elect differently.

LE3. Monotonically Increasing Unique Ballots: If a server s with ballot n is elected as leader by a server p, then all previously elected leaders by p have ballot numbers $m < n$, and the pair (s, n) is unique.

Notice that LE1 does not require non-QC servers to elect a leader. It is not needed as a follower only needs to be connected to the leader to participate in the log replication. Furthermore, LE2 is weaker than a usual agreement property in leader election. Multiple leaders are allowed to elect themselves in different majorities with at least one overlapping server that is not quorum-connected. This is sufficient in Sequence Paxos since LE3 guarantees that one of those leaders will have the maximum ballot. Thus, the overlapping servers will follow (i.e., promise) that maximum leader in Sequence Paxos to provide stable progress.

5.2 Ballot Leader Election

Ballot Leader Election (BLE) is the leader election protocol in Omni-Paxos. It provides resilience against partial connectivity such that Sequence Paxos is guaranteed to make progress as long as a single QC server exists. In BLE, all servers periodically exchange heartbeats with each other. The heartbeat of a server consists of its ballot number and a flag indicating if it is quorum-connected. The heartbeats allow a server to determine two things: (1) Whether itself is quorum-connected and (2) Which of its peers are alive and quorum-connected. With this information, a server can elect a leader that is quorum-connected and has the highest ballot. The elected leader and its ballot number activate the corresponding Sequence Paxos component logic (② in Figure 3b). A failed leader will be detected once its heartbeat is

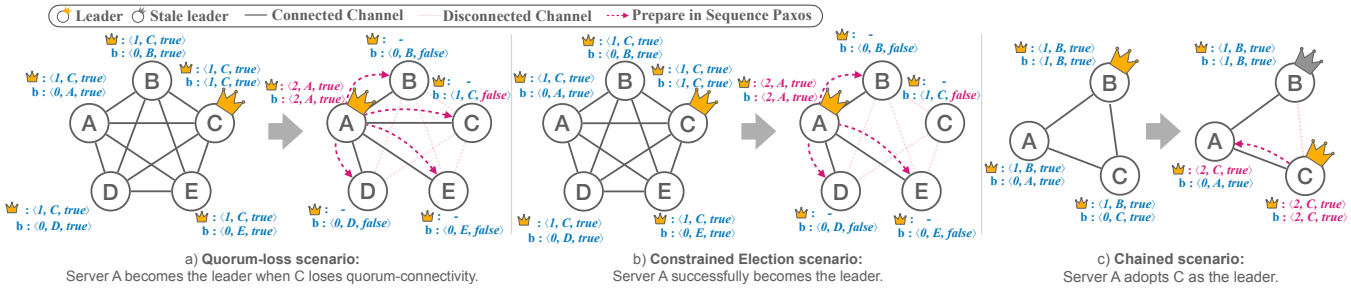


Figure 5. Resilience to partial connectivity in Omni-Paxos.

not received. If the leader is alive but not quorum-connected anymore, the peers will observe that through the corresponding flag in its heartbeats. In both cases, the QC servers will increment their ballot and attempt to become the new leader. The pseudo-code of BLE is shown in Figure 4.

Correctness. To prove the correctness of BLE, we assume that there are some stable periods such that connected servers will learn a period T_{delay} where no late heartbeat is received (as guaranteed by the partially synchronous model). A late heartbeat is simply ignored and does not affect correctness. In practice, T_{delay} can be adaptive.

LE1. As seen in the `checkLeader` function (① in Figure 4), a server can only elect a server that is quorum-connected. Additionally, a server can only perform `checkLeader` and elect a server if a majority of heartbeats were received ②. □

To understand the importance of LE1, we analyze how BLE behaves in the quorum-loss scenario depicted in Figure 5a. When the initial leader C loses its quorum-connectivity, it will notify its peers by sending heartbeats with the *qc* flag *false*. By doing so, C effectively gives up its leadership. When another quorum-connected server (A in this example) observes this, it will increment its ballot to become the leader. We also notice that despite receiving the higher ballot from A, the other servers will not elect A **in BLE** as they are not QC and therefore cannot perform `checkLeader`. Instead, they will promise A when receiving ⟨Prepare⟩ with a higher ballot number in Sequence Paxos, and progress can be made.

LE2. We consider how LE2 is satisfied in every possible case of connectivity between quorum-connected servers. A cluster can have only one quorum-connected server, or multiple quorum-connected servers that are either connected or disconnected from each other.

i) Single QC server. The only QC server eventually receives a majority of heartbeats, and its own will be the only one with the quorum-connected flag *true*. As a result, it elects itself and LE2 is trivially satisfied. □

An example of this is the constrained election scenario shown in Figure 5b. Server A is the only QC server and increments its ballot when it disconnects from C. As in the quorum-loss scenario, $\{B, D, E\}$ will not elect A in BLE as they cannot perform `checkLeader`. Instead, they will promise A in Sequence Paxos. This is an example of only requiring the candidate to be quorum-connected (QC) rather than to be elected by other quorum-connected servers (EQC). Another important observation from this scenario is that the log progress is not a strict requirement in BLE. Even if A has an outdated log, it can get elected and then catch up the missing entries during the Prepare phase of Sequence Paxos.

ii) Multiple connected QC servers. The QC servers will eventually receive the ballots of each other since they are connected. As the ballots are totally-ordered (LE3), they all elect the same server with the highest ballot number in `checkLeader`. □

iii) Multiple disconnected QC servers. If the QC servers are disconnected, then each of them is connected to a majority with at least one overlapping server that is not quorum-connected. In each of these different majorities, either case i or case ii applies and LE2 is thus satisfied. □

Figure 5c shows such an example: All servers are quorum-connected, but B and C are disconnected from each other. After disconnection, C will timeout waiting for B's heartbeats and increment its ballot. Both A and C will elect C with the higher ballot in BLE. After A has promised C in Sequence Paxos, it will not replicate entries from B due to B's lower round number. However, the ballot of A remains unchanged and there is no additional information in the ballots regarding who the current leader is. B will therefore not cause any leader changes and the cluster can progress with $\{A, C\}$.

LE3. As seen in `checkLeader`, the ballots are monotonically increasing since a server only elects a server with a higher ballot than the previous leader. Every ballot $b = (s, pid)$ is unique as pid is the unique identifier of each server ①. □

With the decoupled design of Omni-Paxos, BLE can be customized to improve performance. The ballot can be extended with a custom field c such that $b = (s, c, pid)$. Leader candidates can thus be assigned priorities according to the use case. Note that such extension is only used for breaking ties between ballots and does not affect liveness. An elected candidate must still be quorum-connected.

6 Reconfiguration

We now describe the reconfiguration in Omni-Paxos. Omni-Paxos can complete reconfiguration fast even when newly added servers are disconnected from the old leader. The main differentiation compared to other protocols lies in having a service layer with a cross-configuration scope which allows log migration to execute in a decentralized fashion.

A configuration c_i represents the fixed set of servers running an instance of Sequence Paxos and BLE. To reconfigure, e.g. from $c_i = \{A, B, C\}$ to $c_{i+1} = \{C, D, E\}$, the current configuration must first be stopped via a *stop-sign* (*SS*) entry in the log. The *SS* contains the set of servers in c_{i+1} and gets decided following the normal Sequence Paxos protocol in c_i, with the exception that once *SS* is chosen, no further entries can be decided in c_i. Recall from §3 that the replicated log is stored in the service layer of a server. When *SS* is decided in c_i, the service layer is responsible for starting c_{i+1} safely. If a server s is part of both configurations, then it is safe for the service layer of s to directly start its BLE and Sequence Paxos components of c_{i+1}, as it has already replicated all log entries from c_i. The service layer of s should also notify new servers of c_{i+1}, since they did not participate in c_i and thus did not observe the *SS*. For safety, the service layer of these servers only starts their BLE and Sequence Paxos components of c_{i+1} when the complete log has been fetched from other servers. Note that this log migration only involves decided entries and is performed in the service layer, completely isolated from the underlying log replication.

6.1 Benefits of The Service Layer

The following benefits are introduced by the service layer:
Parallel Log Migration - Performance. Having a service layer separated from log replication allows for a more flexible and parallel reconfiguration where more servers other than the leader can contribute to log migration. This approach can effectively reduce wait time when new servers are added, alleviating heavy data transfer duties imposed on the leader. Figure 6 depicts log migration (a) restricted to the leader within the same log replication instance, vs (b) when performed in the service layer outside log replication. As shown in Figure 6a, if the leader is solely responsible for log migration, it transfers the complete log to every new server $\{C, D, E\}$. With parallel migration in Figure 6b, the new servers receive different log segments in parallel from any server. Since decided entries cannot be retracted, these can even be fetched from servers that have not reached the *SS* in c_i yet, such as the first 100 log entries migrated from A in this example.
Flexible Migration Scheme - Engineering and Resilience. The service layer allows users to apply custom migration schemes that are optimized for specific use cases, such as to reduce cross-data center transmission costs which have shown to be effective when applied to RSMs [41]. This also

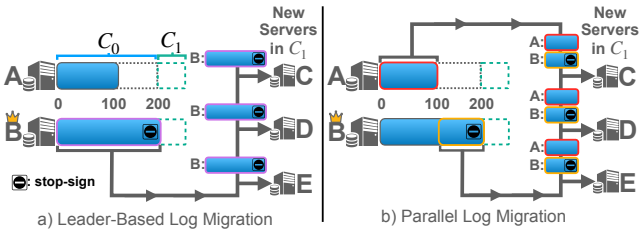


Figure 6. Example of log migration variants.

facilitates for development where migration schemes can be modified in the service layer without affecting the log replication. The parallel migration also improves resilience during reconfiguration. Protocols that perform reconfiguration within the same log replication instance rely on the leader to migrate the log to new servers. Thus, if some server is disconnected from the leader, it cannot complete the reconfiguration. In contrast, in Omni-Paxos, an added server can receive the log from any other server such as an existing follower or even a newly added server that has completed the migration. For example, in Figure 6a, if either C, D, or E disconnect from leader B they cannot catch up the log and start in the new configuration C_1. Thus, if a majority in C_1 consists of new servers that get disconnected from B, the reconfiguration cannot be completed. Whereas, in Omni-Paxos, as long as some server in C_1 manages to collect all log entries from reachable servers in C_0, it can then migrate them to its peers via the service layer.
Isolated Configurations - Engineering. Essentially, the replicated log consists of multiple segments that end with an *SS* entry. Each segment is implemented by an underlying configuration in which BLE and Sequence Paxos components can only communicate with others in the same configuration. Such isolation facilitates for faster development of the system. New versions of the underlying protocols can be deployed in place rather than on the system level of the RSM. Reconfiguration can therefore also be used for software upgrade and not just modifying the set of servers. As the service layer isolates new and old versions in different configurations, this avoids compatibility issues which has been a problem for Raft systems [36]. This design also supports segmented virtualization where different segments of the log can reside in different storage units as in Delos [17].

7 Evaluation

The goal of Omni-Paxos is to provide resilience against any partial network partition and fast reconfiguration while maintaining stable performance. We evaluate Omni-Paxos in each of these aspects through experiments with regular execution (§7.1), partial network partitions (§7.2), and reconfiguration (§7.3).

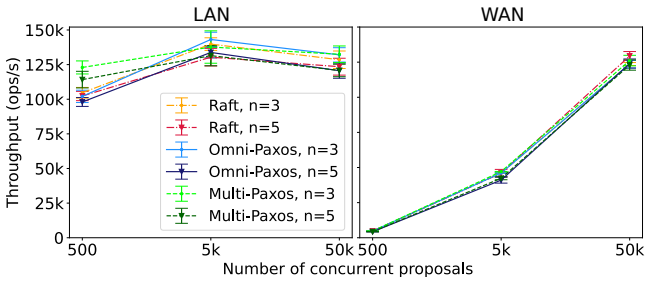


Figure 7. Regular execution with 3 and 5 servers. The error bars show the 95% CI using the t-distribution.

Protocols: We compared Omni-Paxos, Raft, VR, and Multi-Paxos using the distributed actor framework Kompact [13] with TCP. The evaluated protocols are as follows:

- **Omni-Paxos:** The protocols presented in this paper (BLE, Sequence Paxos) was implemented as a Rust library[2].
- **Raft** library written in Rust provided by TiKV [14].
- **Raft PV+CQ:** TiKV's Raft library [14] with `PreVote` and `CheckQuorum` mechanisms [24] enabled.
- **Multi-Paxos:** A Rust version of the Multi-Paxos implementation in frankenpaxos.[3]
- **VR:** An implementation of VR's leader election [31] with Omni-Paxos' log replication. This is only used in the partition experiments to test the the resilience of VR's leader election.

Zab [25, 33] was not included as its leader election properties are a superset of Multi-Paxos, Raft, and VR (see Table 1).

Hardware: The experiments were performed on Google Cloud Compute in the us-central1 region using e2-standard-8 instances with 8 vCPUs and 32 GB memory. The client and each server ran on separate instances. The performance was measured by the client that proposed no-op commands of 8 bytes. The workload level (low-high) of an experiment is set by the parameter *number of concurrent proposals* (*CP*) that determines the number of parallel client requests that are proposed to the RSM. Each experiment lasted for 5 minutes unless stated otherwise and was repeated 10 times. A warmup and cooldown period of 1 minute was performed between each run.

7.1 General Performance

Experiment Description. A cluster of 3 and 5 servers is evaluated in three different workloads *CP* = {500, 5k, 50k}. The cluster was deployed in a LAN setting with RTT=0.2ms and a WAN setting where the RTT from leader to the followers were 105ms (eu-west1) and 145ms (asia-northeast1).

What is the regular performance of Omni-Paxos? Does the Ballot Leader Election incur any performance overhead? As seen in Figure 7, the throughput is similar between Omni-Paxos, Raft, and Multi-Paxos during normal execution. All protocols require a single round trip from the leader to a majority of followers to decide a value, e.g., the `Accept` and `Accepted` in Omni-Paxos. Furthermore, Omni-Paxos' ability to pipeline entries in the Accept phase is shown to perform similarly to Raft and Multi-Paxos. Raft also pipeline entries, while Multi-Paxos decide entries in parallel. However, as the log entries must be contiguous before they can be acknowledged to the client, the performance is similar regardless of whether entries are decided in parallel or through pipelining. Additionally, the overhead of exchanging heartbeats in BLE did not affect Omni-Paxos' performance. From the recorded IO, it could be seen that the BLE overhead is negligible, contributing at most 0.02% of the total IO.

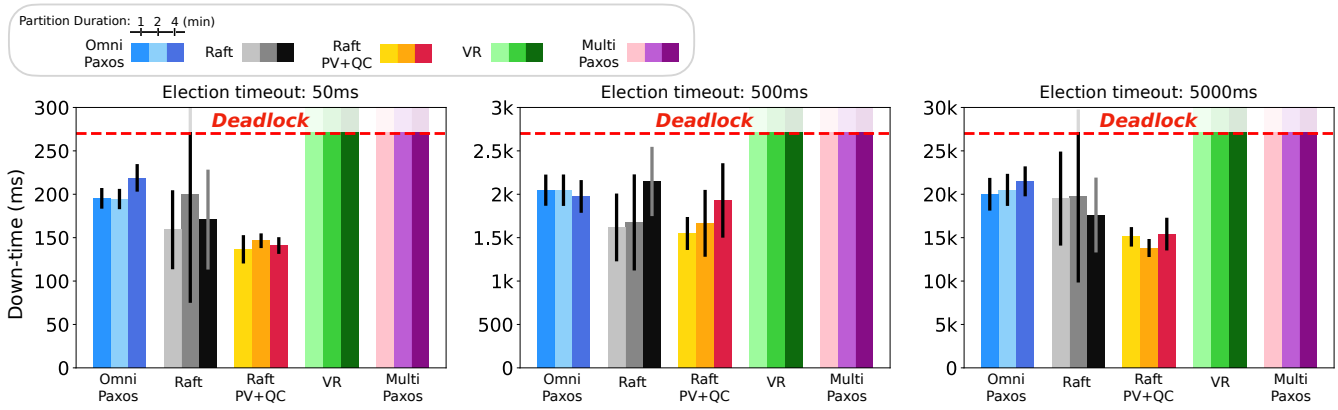
7.2 Resilience to Partial Connectivity

Experiment Description. The client continuously proposes to a cluster that is initially fully connected. After the 1 minute warmup, the partial partitions from §2 are introduced respectively. To ensure that the only quorum-connected server has an outdated log in the constrained election scenario, it is disconnected from the leader earlier. Election timeouts of {50, 500, 50k} ms were tested and the cluster becomes fully connected again after 1, 2, or 4 minutes.

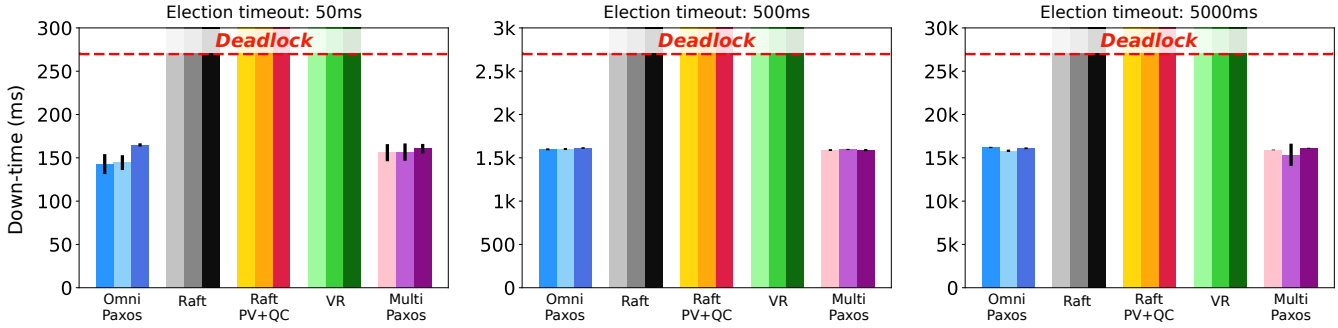
How does Omni-Paxos deal with the quorum-loss scenario? Figure 8a depicts the average down-time, i.e., the duration for when the client received no decided replies. In the quorum-loss scenario, VR and Multi-Paxos do not manage to elect a new leader. VR's requirement of getting elected by a majority of QC servers (EQC) cannot be fulfilled since there is only one QC server (e.g., *A* in Figure 5a). In Multi-Paxos [37], all servers except the QC server will increment their ballot number and attempt to take over leadership. However, as they are not QC, they are not able to get a majority of votes (P1B messages) and subsequently decide new entries. The QC server is the only server that has the potential to gather enough votes but since it still receives heartbeats from the stale leader, it does not suspect any leader failure and therefore does not initiate a leader change at all. As a result, both protocols cannot recover until the partition itself disappears, and have down-times corresponding to the entire duration of the quorum-loss scenario.

Raft managed to elect a new leader and recover but recorded high variance in its down-time duration compared to the rest. This can be attributed to the randomized timers used in Raft which were initially designed to avoid split-votes. However, in this case, they led to repeating term increments by the disconnected followers that prevented the fully-connected server from completing a leader election phase. As a result, we recorded up to 10 term increments from the original term. In contrast, Omni-Paxos recovered with one leader

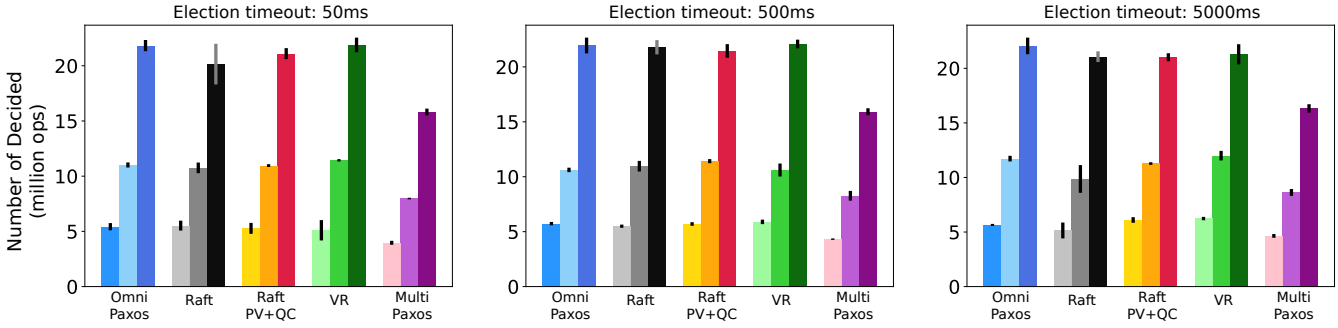
[2] https://github.com/haraldng/omnipaxos
[3] https://github.com/mwhittaker/frankenpaxos



(a) Quorum-loss scenario.



(b) Constrained election scenario.

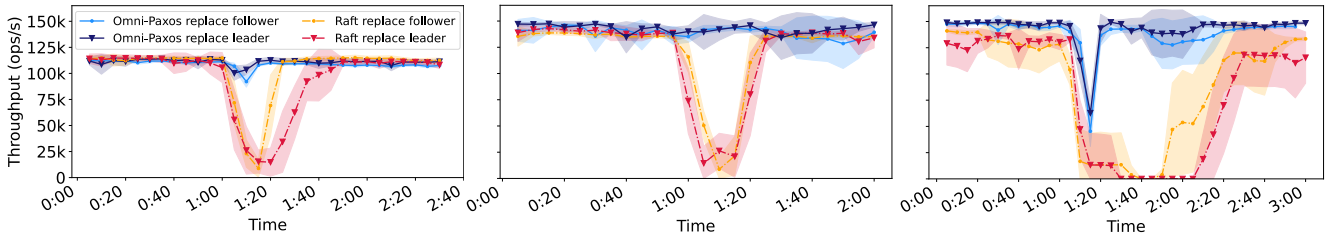


(c) Chained election scenario.

Figure 8. Partial connectivity experiments. The protocols that reach the "deadlock" line in a) and b) have a down-time corresponding to the partition duration. The error bars show the 95% CI using the t-distribution.

change and exhibited an average recovery period of four heartbeat rounds: one round for the leader to detect that it is not quorum-connected, one round to signal that to the QC server, and two subsequent rounds required for the QC server to observe that and then get elected. The variance is introduced by the timing of when the old leader signals that it is not quorum-connected. This could vary as heartbeat rounds overlap between different servers. Raft PV+CQ recovered slightly faster as the leader continuously confirms its status during replication, while Omni-Paxos does it periodically with heartbeats.

How does Omni-Paxos deal with the constrained election scenario? As seen in Figure 8b, VR is still deadlocked for the full duration of the network partition. As in the previous scenario, this is due to only one quorum-connected server exists, and no server can therefore become EQC. Furthermore, Raft and Raft PV+CQ also suffer from down-time corresponding to the partition's duration. As the only QC server in the cluster was disconnected from the leader earlier than the others, it will not have the most updated log and thus fail to get elected. Only Omni-Paxos and Multi-Paxos manage to recover from this scenario. When the leader is suspected of having failed, the QC server will attempt to



(a) Replace single server with $CP = 500$. **(b)** Replace single server with $CP = 50$k. **(c)** Replace majority with $CP = 50$k.

Figure 9. Reconfiguration experiments. The shaded areas show the 95% CI using the t-distribution.

become the leader. Despite having an outdated log, it will succeed as the only candidate requirement is to be quorum-connected. After getting elected, the leader will catch up the log in the prepare phase and progress can then be resumed. In this case, the recorded down-time is constant of 3 timeouts with negligible variance. The down-time is shorter than in the quorum-loss scenario as the QC server directly detects the need for changing leader rather than having to wait for the leader to detect it is not QC and then informing that in an additional heartbeat round.

How does Omni-Paxos handle the chained scenario? Figure 8c shows the number of requests that each protocol managed to decide in the chained partition. Multi-Paxos consistently recorded the lowest throughput among all protocols. This is due to the loop of leader changes caused by gossiping. The loop is not resolved as the fully-connected server, e.g. A in Figure 1a, will not initiate an election since it is connected to the leader regardless of who is currently elected. Multi-Paxos recorded up to 30% worse performance compared to the other protocols when the partitioned lasted for 4 minutes. The other protocols manage to eventually recover from the protocol, and converge to similar performance over time. In general, Raft showed higher variance due to the random leader timeouts where the fully-connected server must timeout and start an election before the other server that is disconnected to the leader. This could require multiple elections and Raft recorded a term that was up to 8 terms higher than the initial value. VR also require the middle server to be elected. If that server is not next in line to be the leader according to VR's pre-determined round-robin schedule, an additional leader change will occur. Thus, the variance of VR is due to it occasionally changing leader twice. Omni-Paxos performs a single leader change (shown in Figure 5c) when the partition occurs and has a more stable performance compared to Raft, Multi-Paxos, and VR. Raft PV+CQ recorded no leader changes as the `PreVote` phase implies that A does not vote for another server as long as its current leader is alive.

Summary. Omni-Paxos is the only protocol that consistently manages to maintain stable progress across all partition scenarios. Other protocols can recover from specific scenarios but suffer in other scenarios from livelock with repeated elections or deadlock with no leader elected at all. Omni-Paxos recovers from all the scenarios in constant time with a single leader change.

7.3 Reconfiguration Speed

Experiment Description. The client proposes a reconfiguration to a cluster of 5 servers under three different workloads $CP = \{500, 5\text{k}, 50\text{k}\}$. To simulate that the cluster has been running for a period of time, the servers are initialized with a log containing 5 million entries. The client warms up by deciding 10 million entries and then proposes a reconfiguration to replace either one server or a majority of servers. Using 8 bytes entries, a new node needs to catch up $(5 + 10) * 8 = 120$MB of data. Only Raft and Omni-Paxos are used in this experiment as the other implementations do not support reconfiguration.

How does Omni-Paxos improve reconfiguration? The parallel log migration in the service layer of Omni-Paxos prevents the leader from becoming a bottleneck. For the single reconfiguration experiment, rather than having the leader transfer 120MB to the new server, the leader and the three continued followers split up the work and transfer 30MB each in parallel. As seen in Figure 9a and 9b, this results in a significantly smaller drop in throughput with a shorter duration compared to Raft. In both workloads, Raft recorded up to 90% lower throughput over a period of 55s compared to 20% and 15s for Omni-Paxos. The case of excessive overloading at the Raft leader can also be inferred by the outgoing traffic volume. The peak IO for the leader over a 5s-window was 109MB compared to 30MB in Omni-Paxos. From Figure 9b, we also observe no clear drop in throughput for Omni-Paxos with 50k concurrent proposals. With a larger pipeline, more proposals can be buffered and instantly be proposed in a large batch when the new configuration starts. The down-time from switching configurations is thus masked behind the net throughput over 5s-windows.

As seen in Figure 9c, replacing a majority affected both protocols more as at least one of the new servers must receive the complete log before c_{i+1} can start. With only two servers of c_i continuing in c_{i+1}, the parallel log migration in

Omni-Paxos was less efficient compared to the previous experiment. These servers had to transfer 60MB to every new server, and a peak of 180MB of outgoing data was recorded for one server. As a result, Omni-Paxos showed 80% lower throughput for a period of 15s. However, it still outperformed Raft significantly. Raft recorded up to 40s of complete down-time and took up to 120s to recover the performance levels prior reconfiguration. The leader bottleneck became more prominent in this experiment with a peak of 336MB of outgoing data. Raft also suffered from unexpected leader changes which required multiple attempts to complete the reconfiguration. In some runs, it was not the initial leader who committed the reconfiguration. The recorded outgoing data indicates that both leaders performed log migration, with the initial leader doing most of the work. This suggests that it got too overloaded, leading to another server getting elected. The new leader could then complete the reconfiguration as large parts of the log had already been transferred by the previous leader.

8 Discussion and Future Work

QC resilience vs. Fully-connected resilience. While the partial connectivity problem has been gaining traction [24, 30], our work establishes a new minimum requirement for achieving progress, from the classic fully-connected majority quorum down to a single quorum-connected server. As shown by the evaluation, this effectively increases resilience, which is the main objective of RSMs, without affecting performance. Supporting a QC leader is also a design choice for stable performance since a QC server could remain as the leader even if there are servers with better connectivity (e.g., server A in Figure 5c). On the other hand, optimizing for the most well-connected leader must be carefully considered as servers are allowed to take over leadership even when the RSM is stable. This could introduce leader changes and down-time whenever the network changes. We argue that stability is crucial for RSMs operating in the dynamic and highly virtualized cloud environments of today. A possible optimization in Omni-Paxos is to include connectivity data in the ballot of BLE, such that the better connected servers are prioritized when a leader change is really required.

Half-duplex partial connectivity: The Quorum-connected Leader Election properties (§5.1) can be extended to support half-duplex links where communication can only be made in one direction. To provide liveness, the leader must still be quorum-connected with full-duplex links, which is what BLE elects by default using the heartbeat request and response.

9 Related Work

Sequence Consensus. The safety properties for replicating a strictly growing log without gaps were first introduced in the context of Generalized Consensus [26]. However, no Sequence Consensus protocol was presented in that work. The simplicity of replicating a growing log has resulted in wide adoption of Sequence Consensus protocols. The most popular open-source systems [6, 8, 23, 36, 41] and libraries [4, 5, 7, 14] are based on Sequence Consensus protocols such as Raft [34], VR [31], and Zab [25, 33]. Similar to Omni-Paxos, these protocols are self-contained and support leader election and reconfiguration. However, they are not resilient to partial connectivity due to their monolithic design as described in §2. Furthermore, the tight-coupling of components in these protocols also affects their reconfiguration performance. The reconfiguration of Zab [35] is similar to Raft where it is tightly coupled with log replication and uses leader-based log migration. VR takes a similar approach as Omni-Paxos, using the concept of a stop entry from Stoppable Paxos [29]. However, VR integrates reconfiguration with log replication which introduces complexity and state that are unrelated to the log replication protocol.

Multi-Paxos [27, 37]. Similar to the described Sequence Consensus protocols, Multi-Paxos is also susceptible to partial connectivity. Multi-Paxos differs instead from Sequence Consensus by deciding entries independently to form a log. However, the contrasting designs do not constitute for any differences in performance (see §7.1). In both approaches, an entry is decided in a single round-trip and the leader can start replicating new entries before the preceding one has been decided. Furthermore, the log must in any case be contiguous without gaps before the entries can be executed by the RSM. Thus, there is no performance difference between deciding entries independently and deciding in a strictly growing log with pipeline parallelism. However, deciding in parallel requires bookkeeping to track gaps and keep metadata per entry. Avoiding this additional complexity was also a main motivation behind Raft [34].

Overlay networks. A different approach to tackle partial connectivity is through overlay networks. Nifty [15] is a transparent network layer that builds an overlay between nodes that route packets to mask partial connectivity. While this has the advantage of being a general solution, it could however result in both reduced fault-tolerance and performance when applied to RSMs. Instead of tolerating up to a majority of failures, any failure in the dissemination path can cause down-time. Furthermore, since partial connectivity is masked, the elected leader might not actually be quorum-connected. As a result, the RSM could suffer from higher commit latency due to more network hops. Omni-Paxos is designed to be aware of partial connectivity and do not require an additional overlay network.

10 Conclusion

Partial connectivity causes existing RSM protocols to livelock with repeated leader changes or deadlock by failing to elect a leader at all. We address these problems by presenting Omni-Paxos, a complete RSM system that is consistently resilient to

partial connectivity. Omni-Paxos tackles the problem of partial connectivity by (1) decoupling leader election from log replication to remove unnecessary requirements or gossiping information unrelated to leader election, and (2) integrating connectivity to elect a leader that is quorum-connected. This allows Omni-Paxos to boost the resilience of RSMs down to only requiring a single quorum-connected server to make progress. As shown in the evaluation, Omni-Paxos recovers from any partial network partition in constant time. Furthermore, the decoupled design of Omni-Paxos provides additional benefits such as significantly shorter reconfiguration periods through parallel log migration.

Acknowledgments

This work has been supported by the Swedish Foundation of Strategic Research (Grant No.: BD15-0006), Google Cloud Research Credits Program, and Wallenberg AI NEST (Data-Bound Computing). Furthermore, we would like to thank Lars Kroll for early-stage contributions to the Sequence Paxos specification, as well as peer reviewers and Marios Kogias for the improvement suggestions.

References

[1] 2010. *MapReduce-using map output fetch failures to blacklist nodes is problematic*. Retrieved 2023-03-14 from https://issues.apache.org/jira/browse/MAPREDUCE-1800

[2] 2014. *ElasticSearch-Partial network partition and retries*. Retrieved 2023-03-14 from https://github.com/elastic/elasticsearch/issues/6105

[3] 2015. *ElasticSearch-cluster broken after switches upgrade*. Retrieved 2023-03-14 from https://github.com/elastic/elasticsearch/issues/9495

[4] 2022. *baidu/braft*. Retrieved 2022-12-15 from https://github.com/baidu/braft

[5] 2022. *Dragonboat - A Multi-Group Raft library in Go*. Retrieved 2022-12-15 from https://github.com/lni/dragonboat

[6] 2022. *etcd*. Retrieved 2022-12-15 from https://etcd.io/

[7] 2022. *hashicorp/raft*. Retrieved 2022-12-15 from https://github.com/hashicorp/raft

[8] 2022. *TiKV*. Retrieved 2022-12-15 from https://tikv.org/

[9] 2023. *Apache Flink*. Retrieved 2023-03-14 from https://flink.apache.org

[10] 2023. *Apache Kafka*. Retrieved 2023-03-14 from https://kafka.apache.org

[11] 2023. *Apache Spark*. Retrieved 2023-03-14 from https://spark.apache.org

[12] 2023. *Docker Engine: Raft Consensus in Swarm Model*. Retrieved 2023-03-14 from https://docs.docker.com/engine/swarm/raft/

[13] 2023. *The Kompact Actor Framework*. Retrieved 2023-03-14 from https://github.com/kompics/kompact

[14] 2023. *TiKV Raft library in Rust*. Retrieved 2023-03-14 from https://github.com/tikv/raft-rs

[15] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. 2020. Toward a Generic Fault Tolerance Technique for Partial Network Partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 351–368. https://www.usenix.org/conference/osdi20/presentation/alfatafta

[16] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An Analysis of Network-Partitioning Failures in Cloud Systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 51–68. https://www.usenix.org/conference/osdi18/presentation/alquraan

[17] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, et al. 2020. Virtual Consensus in Delos. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 617–632.

[18] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 335–350.

[19] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. 1996. The Weakest Failure Detector for Solving Consensus. *J. ACM* 43, 4 (July 1996), 685–722. https://doi.org/10.1145/234533.234549

[20] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[21] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. 2020. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 325–338.

[22] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[23] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX annual technical conference*, Vol. 8.

[24] Chris Jensen, Heidi Howard, and Richard Mortier. 2021. Examining Raft's behaviour during partial network failures. In *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems*. 11–17.

[25] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*. 245–256. https://doi.org/10.1109/DSN.2011.5958223

[26] Leslie Lamport. 2005. Generalized consensus and Paxos. (2005).

[27] Leslie Lamport. 2019. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*. 277–317.

[28] Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

[29] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2008. Stoppable paxos. *TechReport, Microsoft Research* (2008).

[30] Tom Lianza and Chris Snook. 2020. *A Byzantine failure in the real world*. Retrieved 2023-03-14 from https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/

[31] Barbara Liskov and James Cowling. 2012. Viewstamped replication revisited. (2012).

[32] Dahlia Malkhi, Mahesh Balakrishnan, John D Davis, Vijayan Prabhakaran, and Ted Wobber. 2012. From Paxos to CORFU: a flash-speed shared log. *ACM SIGOPS Operating Systems Review* 46, 1 (2012), 47–51.

[33] André Medeiros. 2012. ZooKeeper's atomic broadcast protocol: Theory and practice. *Aalto University School of Science* (2012).

[34] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[35] Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio P. Junqueira. 2012. Dynamic Reconfiguration of Primary/Backup Clusters. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 425–437. https://www.usenix.org/conference/atc12/technical-sessions/presentation/shraer

[36] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin,

Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.
[37] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos made moderately complex. *ACM Computing Surveys (CSUR)* 47, 3 (2015), 1–36.
[38] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–17.
[39] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. 2020. Scaling Replicated State Machines with Compartmentalization [Technical Report]. *arXiv preprint arXiv:2012.15762* (2020).
[40] Michael Whittaker, Neil Giridharan, Adriana Szekeres, Joseph Hellerstein, Heidi Howard, Faisal Nawab, and Ion Stoica. 2021. Solution: Donut Paxos: A Reconfigurable Consensus Protocol. In *Journal of Systems Research-Mar 2021*.
[41] Siyuan Zhou and Shuai Mu. 2021. Fault-Tolerant Replication with Pull-Based Consensus in MongoDB. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 687–703. https://www.usenix.org/conference/nsdi21/presentation/zhou

A Sequence Paxos: Proof of Safety

SC1. Validity: *If a server decides on a log L then L only contains proposed commands.*

By inspection of the pseudo-code shown in Figure 3b, any decided command must come from either the `log` or the `buffer` of some server ①. And as seen in ⑥, the `buffer` and the `log` contains only commands from clients. Furthermore, the properties of the session-based FIFO perfect link do not invent any new commands. □

SC3. Integrity: *If a server decides on a log L and later decides on L' then L is a strict prefix of L'.*

The invariant P2c is central for the proof of SC3. Recall from §4.2 the invariant P2c: *For any v and n, if a proposal with sequence v and number n is issued, then there is a majority-set S of acceptors such that sequence v is an extension to the sequence of the highest-numbered proposal less than n accepted by the acceptors in S*. Sequence Paxos maintains P2c with log synchronization in the Prepare phase. Any new leader must first collect a majority of promises to adopt the most updated log, which is guaranteed to include any chosen sequence. Only after adopting the most updated log and synchronizing it with the promised followers can a leader propose new commands that extend the log.

Proof by induction.
We are going to prove the property SC3 using induction on the sequence of decisions by any server. We denote the i^{th} decision by a server as $SC3(i)$.

Base case: Show that $SC3(1)$ holds, that is, the property holds on the first decision of a server.

Initially, all servers have an empty log. And any log is an extension to the empty log. Thus, if any servers decide it will be an extension to the empty log and the base case is satisfied.

Inductive step: We want to show that for any server s if $SC3(i)$ holds for any $i < j$, then $SC3(j)$ also holds for s. By the induction hypothesis for a particular $i < j$, $SC3(i)$ holds. Furthermore, we assume that log L was the i^{th} decision of a server s. We perform a case analysis on the role of s:
(1) Leader. We assume s is the leader of a round n. Since L is already decided in a lower round, it is guaranteed that at least one server among any majority has L as a prefix. In the Prepare phase ④, s adopts the log with the highest round number, denoted by L_{max}. Note that L_{max} includes the latest chosen log, since it is the most updated log in the majority. Given the invariant P2c, it is guaranteed that L_{max} must have L as a prefix. The leader s adopts L_{max} and extends it. Thus, any subsequent decision j by s in this round extends L_{max}, which in turn extends L and $SC3(j)$ holds.
(2) Follower in the Prepare phase. We assume s is a follower in round n and has received a ⟨Prepare⟩ message with n from the leader. Since L is decided by s, the suffix in the ⟨Promise⟩ sent by s is guaranteed to include any possibly missing entries of L at the leader. Given this and the invariant P2c, the adopted log L_{max} at the leader must have L as a prefix. L_{max} is then adopted by s when handling ⟨AcceptSync⟩ ⑤. Given the FIFO property of the communication, any later accepted entries extends L_{max}. Thus, any subsequent decision j by s is an extension of L and $SC3(j)$ holds.
(3) Late follower after the Prepare phase. We assume s is a follower in round n and has received a ⟨Prepare⟩ message with n from the leader. However, now we assume that the leader has already completed the Prepare phase in round n. In this case, L is already a prefix at the leader. Thus, what is sent in the ⟨AcceptSync⟩ and the following ⟨Accept⟩ messages is guaranteed to extend L at s. Any subsequent decision j by s is an extension of L and $SC3(j)$ holds.

The property holds for all possible roles of a server, establishing the inductive step. □

SC2. Uniform Agreement: *For any two servers that decided logs L and L' respectively then one is the prefix of the other.*

Proof by contradiction.
Assume that L is not a prefix of L' and vice versa. Since both logs are decided, there are two majorities Q and Q' that have decided L and L' respectively. As any two majorities intersect, there must be at least a server $s \in (Q \cap Q')$ that has decided both L and L'. However, as shown by the proof of SC3, s can only decide on a log incrementally. Thus, L must be a prefix of L' or vice versa, a contradiction. □

B Artifact Appendix

B.1 Abstract

This artifact provides a description to reproduce the benchmarks used for the evaluation. The code is made publicly available on GitHub with detailed instructions on how to setup, run, and reproduce the results as they appear in Section 7 of the paper.

B.2 Description & Requirements

B.2.1 How to access. The artifact is available on GitHub: https://github.com/haraldng/omnipaxos-artifacts or via the DOI: https://doi.org/10.5281/zenodo.7737776.

B.2.2 Hardware dependencies. Our evaluation was performed on Google Cloud Compute using nine VM instances of `e2-standard-8`, each with 8 vCPUs, 32 GB memory, and 32GB boot disk size.

B.2.3 Software dependencies. The environment in which we performed our evaluation was as follows:

- Ubuntu 18.04
- Rust: rustc 1.53.0-nightly (3709ae324 2021-04-25)
- JDK: openjdk 11.0.11 2021-04-20
- Scala 2.13.0
- SBT 1.6.2
- Ammonite 2.3.8
- protobuf 3.6 (in particular protoc)

B.2.4 Benchmarks. The repository is self-contained and has all benchmarks required to reproduce the results.

B.3 Set-up

We refer evaluators to the `README.md` in the repository that describes in detail how to install all required dependencies and set up the VM instances to reproduce the results.

B.4 Evaluation workflow

B.4.1 Major Claims.

- **C1: Omni-Paxos is the only protocol that can recover from all the described partial connectivity scenarios** (constrained, quorum loss, and chained scenario). The other protocols either deadlock or/and livelock in at least one scenario. This is proven by the experiment (E1) described in §7.2 whose results are illustrated in Figure 7.
- **C2: The regular performance of Omni-Paxos is on par with existing state-of-the-art protocols in both LAN and WAN settings**. This is proven by the experiments (E2-E4) described in §7.1 whose results are illustrated in Figure 8.
- **C3: Omni-Paxos has a lower reconfiguration overhead than Raft**. When a single server is replaced, there is a drop in throughput of at most 20% lasting up to 15s for Omni-Paxos, compared to 90% and 55s for Raft. When replacing a majority of servers, Omni-Paxos records at most 80% lower throughput but recovers to regular performance after 15s. Raft is completely down for up to 40s and takes up to 120s to recover. This is proven by the experiments (E5) described in §7.3 whose results are illustrated in Figure 9.

B.4.2 Experiments. We use five experiments to evaluate the claims (C1-C3). Under the "Results" header of each experiment description, we specify the output location of the results. This should be used for plotting the corresponding Figures in `visualisation/Plotting.ipynb`.

E1: Partial Connectivity [15 human-minutes + 98 compute-hour]: This experiment simulates the partial connectivity scenarios described in §2 and is used to evaluate claim (C1).

[Preparation]

1. Set up the VM instances according to the description in the Setup Guide of `README.md`. This experiment requires 6 instances: 1 master + 5 server instances, all in the same region.
2. On the master VM instance: change branch to `pc` and build using the command `./build.sc`

[Execution]

On the master instance, run:
`./bench.sc remote --runName pc`

[Results]

The results will be written to the following directory of the master:
`meta_results/pc`

The plotted figure should show that Omni-Paxos, Raft, and Raft PV+QC can recover from the quorum-loss scenario, while VR and Multi-Paxos are deadlocked. In the constrained scenario Omni-Paxos and Multi-Paxos can recover while the other protocols are deadlocked. In the chained scenario, Multi-Paxos have lower number of decided proposals for all timeouts and scenarios due to repeated leader changes. These results should thus indicate that only Omni-Paxos can tolerate all the scenarios as claimed in (C1).

E2: Normal LAN [15 human-minutes + 22 compute-hour]: This experiment measures performance in normal scenarios without any partitions or leader changes and is used to evaluate claim (C2).
[Preparation]

Same as E1, but use the branch `normal-lan` in step 2.

[Execution]

On the master instance, run:
`./bench.sc remote --runName normal-lan`

[Results]

The results will be located in:
`meta_results/normal-lan`

This directory should contain one sub-folder for every experiment (e.g., `5-500-off`), each containing a `num_decided` folder with the number of decided proposals per protocol. The plots (see section Plotting in `README.md`), should support the claim (C2) by showing that Omni-Paxos, Raft, and Multi-Paxos have similar throughput with overlapping CI intervals.

E3: Normal WAN 3 servers [15 human-minutes + 11 compute-hour]: Same as E2, but with only three servers that has high latency (> 100ms) to each other.
[Preparation]

1. Set up the VM instances according to the description in the Setup Guide of `README.md`. Ensure the VMs are in the following regions: Server 1 located in asia-northeast1, server 2 in europe-west1, the master and server 3 in us-central1. Set `nodes.conf` to assign the server with the correct id (e.g., the server in us-central is assigned to id 3).
2. Same as step 2 in E1, but use the branch `normal-wan3`

[Execution]

On the master instance, run:
`./bench.sc remote --runName normal-wan3`

[Results]

Same as E2 but the results are found in:
`meta_results/normal-wan3`

E4: Normal WAN 5 servers [15 human-minutes + 11 compute-hour]: Same as E3 but with five servers where the latency between leader and followers is high.
[Preparation]

1. Set up the VM instances according to the description in the Setup Guide of `README.md`. Ensure the VMs are in the following regions: Server 1 and 2 located in asia-northeast1, server 3 and 4 in europe-west1, the master and server 5 in us-central1. Set `nodes.conf` accordingly.
2. Same as step 2 in E1, but use the branch `normal-wan5`

[Execution] On the master instance, use the command:
./bench.sc remote --runName normal-wan5

[Results] Same as E2 but the results are found in:
`meta_results/normal-wan5`

E5: Reconfiguration [15 human-minutes + 11 compute-hour]: This experiment performs reconfiguration to replace a single or a majority of servers and is used to evaluate claim (C3).
[Preparation]

1. Set up the VM instances according to the description in the Setup Guide of `README.md`. This experiment requires 9 instances: 1 master + 8 server instances, all in the same region.
2. Same as step 2 in E1, but use the branch `reconfig`.

[Execution]

On the master instance, use the command:
`./bench.sc remote --runName reconfig`

[Results]

The results will be located in:
`meta_results/reconfig`

Each experiment folder will have a sub-folder `windowed` with files that have recorded the number of decided proposals for every 5s window. When plotted, we see the throughput over time. The plots should show that Omni-Paxos has a smaller drop over a shorter period of time similar to the values specified in (C3). This should support the claim (C3) that Omni-Paxos has lower reconfiguration overhead with smaller impact on throughput and faster recovery compared to Raft.

B.5 General Notes

Please see `MISC.md` for useful tips and tools when reproducing the benchmarks.