SwiftPaxos: Fast Geo-Replicated State Machines

Fedor Ryabinin, *IMDEA Software Institute and Universidad Politécnica de Madrid;* Alexey Gotsman, *IMDEA Software Institute;* Pierre Sutra, *Télécom SudParis and INRIA*

https://www.usenix.org/conference/nsdi24/presentation/ryabinin

This paper is included in the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

SwiftPaxos: Fast Geo-Replicated State Machines

Fedor Ryabinin
IMDEA Software Institute
Universidad Politécnica de Madrid

Alexey Gotsman
IMDEA Software Institute

Pierre Sutra
Télécom SudParis
INRIA

Abstract

Cloud services improve their availability by replicating data across sites in different geographical regions. A variety of state-machine replication protocols have been proposed for this setting that reduce the latency under workloads with low contention. However, when contention increases, these protocols may deliver lower performance than Paxos. This paper introduces SwiftPaxos—a protocol that lowers the best-case latency in comparison to Paxos without hurting the worst-case one. SwiftPaxos executes a command in 2 message delays if there is no contention, and in 3 message delays otherwise. To achieve this, the protocol allows replicas to vote on the order in which they receive state-machine commands. Differently from previous protocols, SwiftPaxos permits a replica to vote twice: first for its own ordering proposal, and then to follow the leader. This mechanism avoids restarting the voting process when a disagreement occurs among replicas, saving computation time and message delays. Our evaluation shows that the throughput of SwiftPaxos is up to 2.9x better than state-of-the-art alternatives.

1 Introduction

Context. Today's cloud services run in data centers scattered around the world. The critical part of these services is replicated at different geographical sites and maintained strongly consistent [9, 14, 40, 48]. To achieve this, cloud providers rely on *state-machine replication (SMR)* [45], where a service is defined by a deterministic state machine and each site maintains its own replica of the machine. An SMR protocol coordinates the execution of commands at the sites, ensuring that they stay in sync. The resulting system is *linearizable* [21], providing an illusion that each command applied to the service executes instantaneously at all sites.

Unfortunately, common SMR protocols, such as Paxos [29] and Raft [39], have a high latency in geo-replicated deployments. These protocols funnel all commands through a *leader* site, which orders commands and persists them at replicas. If the leader fails, a new one is elected; the period when a given site acts as a leader is called a *ballot*. In such protocols the client finds out the result of a command execution after 4 message delays: a round trip from the client to the leader plus a round trip from the leader to the replicas. This high latency is made worse by the fact that geo-replicated transaction processing systems (such as Spanner [14]) use Paxos multiple times when executing a single transaction, e.g., to implement a fault-tolerant version of two-phase commit.

Problem. There have been several proposals of SMR protocols aiming to lower latency. One approach, pioneered by Fast Paxos [31], is for clients to contact replicas directly, bypassing the leader, and let each replica order the command independently. If enough replicas (usually $> 3/4$ of the total) agree on the ordering of the command, it takes the *fast path*. Otherwise, it is processed via a *slow path*, which requires extra message exchanges to resolve the disagreement. We can increase the chances of spontaneous agreement on command ordering by observing that, to satisfy linearizability, it is enough that replicas only agree on the order of non-commuting (aka *conflicting*) commands [30, 42]. This allows taking the fast path when there is no contention, i.e., when replicas receive conflicting commands in the same order, as is often the case in application workloads [26, 38, 41]. Unfortunately, protocols using this approach, such as Generalized Paxos [30], become very expensive once even a single pair of commands contend: in this case the protocol changes the ballot, similarly to how it would handle a leader failure. This requires transferring a large amount of state between replicas and disrupts the processing of all commands, even those that do not conflict. For this reason, Generalized Paxos has not been used in practice.

An alternative approach eliminates the leader altogether, allowing the replicas to order commands in a peer-to-peer manner (EPaxos [38] and follow-ups [3, 6, 17]). To perform optimally, such leaderless protocols require clients to be co-located with a data replica. However, as replication is expensive, geo-distributed systems often do not maintain a data replica at each client site [9, 40]. In addition, the execution of commands in this class of SMR protocols is often delayed by a form of a convoy effect. As a consequence, these protocols have a high tail latency even at low conflict rates; once the conflict rate increases, the latency skyrockets [6, 19, 50].

Contributions. To address these limitations, in this paper we present SwiftPaxos—a new SMR protocol providing fast linearizable operations over geo-replicated data. It processes a command in 2 message delays when this command commutes with concurrently submitted ones (fast path). Otherwise, SwiftPaxos takes just one extra delay for the leader to resolve the conflict, requiring 3 message delays overall (slow path). Unlike in Generalized Paxos [30], the slow path does not disrupt the system and does not require transferring significant amounts of state. Since in SwiftPaxos conflicts are resolved by the leader, it also exhibits low tail latency, unlike EPaxos [38]. Thus, SwiftPaxos lowers the best-case latency in comparison to Paxos without hurting the worst-case one.

To achieve such benefits, in SwiftPaxos a client sends its command c directly to the replicas, which compute the set of c's *dependencies*—commands conflicting with c that should be executed before it. At each replica, dependencies form a partial order on commands, dictating how they should be applied to the local copy of the state machine. To compute the dependencies of a command, replicas make *proposals* based on the order in which they receive commands, and then agree on one of these. If enough replicas make the same proposal, the command is processed in a single round-trip (fast path). Otherwise, the replicas adopt the leader's proposal, requiring one extra message delay (slow path).

The key novelty of this scheme is that on the slow path a replica votes for two different proposals: first for its own and then for the leader's, with latter superseding the former. Such double voting would usually be unsafe in Paxos-like SMR protocols: this is why Generalized Paxos [30] can only resolve a disagreement by changing the ballot. It is safe in SwiftPaxos because we include the leader into all fast quorums. Hence, if a fast quorum replica disagreed with the leader, the replica knows that the command could not have been committed on the fast path. The replica can then quickly correct its vote without compromising safety by sending a new message overruling the previous one. This enables processing a command on the slow path in just 3 message delays.

We have experimentally evaluated SwiftPaxos across 13 regions on Amazon EC2. Depending on contention, the protocol delivers 16–29% lower average latency than Paxos, and in mixed YCSB workloads [13] its throughput is up to 2.9x higher than EPaxos.

2 System Model

We consider a geo-distributed message-passing system where processes may fail by crashing, but do not behave maliciously. The processes are split into *replicas*, running a distributed service, and *clients* using it. We denote the set of replicas by $\mathcal{R}$ and assume that there are $N = 2f+1$ of these, at most f of which may fail. Each process in $\mathcal{R}$ models a server located in a separate data center; clients can be co-located with servers or located elsewhere. The set of replicas can be changed using standard reconfiguration techniques [29, 46], and we omit details related to this.

State-machine replication (SMR) is a common approach to designing highly available distributed services in the above system [14, 45]. The service is defined by a deterministic state machine, which has a set of states $\mathcal{S}$ and accepts a set of *commands* $\mathcal{C}$. Given a command c and a state s, a function $\texttt{exec}(c,s)$ returns a pair (r,s') of the return value r and the new state s' obtained by executing c in s. Each replica maintains its own copy of the state machine, accessible via a variable $\texttt{state}$. An *SMR protocol* coordinates the execution of commands at the replicas, ensuring that their copies of the state machine stay in sync.

Clients are stateless, but they know how to contact the replicas. The SMR protocol allows a client to submit a command c for execution using an API call $\texttt{submit}(c)$. Each submitted command c is tagged with a unique identifier $\texttt{id}(c)$. A function $\texttt{client}$ associates each identifier with the client that submitted the command. When the SMR protocol obtains the response value r of a command c, it upcalls into the client with a notification $\texttt{response}(\texttt{id}(c), r)$.

The protocol we propose in this paper satisfies *linearizability* [21]. Informally, this means that commands appear to clients as if executed sequentially on a single copy of the state machine in an order consistent with the *real-time order*, i.e., the order of non-overlapping command invocations. To satisfy linearizability, it is enough that replicas agree on the execution order of non-commuting commands [30, 42]. More precisely, two commands c and d *commute* if for every state s of the state machine: *(i)* executing c followed by d or d followed by c in s leads to the same state; and *(ii)* c returns the same response in s as in the state obtained by executing d from s, and vice versa. When the two commands do not commute, we say that they *conflict*, written $c \bowtie d$. Conflicts can be over-approximated using the service API: e.g., in a key-value store operations on different keys commute. In transactional systems with deferred update replication, such as Spanner [14], conflicts can be detected at commit time.

To give a specification of an SMR protocol, we first define a relation $\prec$ over $\mathcal{C}$ so that $c \prec d$ if: *(i)* $c \bowtie d$ and some replica p executes c before d; or *(ii)* c was executed by some replica before any client submitted d. An SMR protocol needs to satisfy the following properties:

Validity. If a replica executes a command c, then some client has submitted c before.

Integrity. A replica executes each command at most once.

Ordering. The relation $\prec$ is acyclic.

Nontriviality. The return value r obtained by the client for a command c is the result of c's execution at some replica.

Liveness. If a command c is submitted by a non-faulty client or executed by some replica, then c is eventually executed by all non-faulty replicas.

Implementing this specification is sufficient to ensure the linearizability and responsiveness of the service [30, 42]. In particular, Ordering guarantees that different replicas cannot execute conflicting commands in different orders. As is standard, to ensure liveness we assume that the system is *eventually synchronous* [16], so that message delays between non-failed processes are eventually bounded.

3 Core Concepts and Protocol Overview

We first provide an overview of SwiftPaxos, and then cover it in detail. We have rigorously proved SwiftPaxos correct, but due to space constraints we defer this proof to §A. Instead, in our explanations we state key protocol invariants and informally explain why they hold.

3.1 Ballots

As usual for Paxos-like protocols, SwiftPaxos's execution is divided into a sequence of *ballots*. A replica can be in a single ballot at a time, tracked in a variable bal. Each ballot b has a fixed *leader* replica $\mathsf{leader}(b) = p_{(b \bmod N)}$; all other participants of b are *followers*. If $\mathsf{leader}(b)$ is suspected of failure, a follower initiates a *recovery* procedure, which switches to a higher ballot with a new leader. A variable status at a replica records whether it is operating normally (NORMAL) or is recovering (RECOVERING).

We call a majority of replicas a *quorum*. Each ballot b is associated with a set of *fast quorums* $\mathcal{FQ}(b)$ and a set of *slow quorums* $\mathcal{SQ}(b)$; these are respectively used on the fast and slow paths of SwiftPaxos. For a replica p, we write $\mathit{fast}(p,b)$ and $\mathit{slow}(p,b)$ if p belongs to a fast and a slow quorum of b, respectively. We require that the leader of b belong to every fast and slow quorum of b. Apart from this constraint, slow quorums can be any majorities of replicas. Fast quorums must satisfy a stricter condition (as in Fast Paxos [31]), requiring that any two fast quorums intersect in a majority:

$$\forall Q_1, Q_2 \in \mathcal{FQ}(b).\, |Q_1 \cap Q_2| > N/2. \qquad \text{(FQI)}$$

This condition can be satisfied in different ways. In the rest of the paper we consider the following two:

(C1) A fast quorum is any set containing $> 3/4$ of all replicas, including the leader.

(C2) There is a unique fast quorum consisting of a fixed majority of replicas, including the leader.

These two quorum systems offer different optima in the trade-off defined by (FQI). On the one hand, (C1) supports up to $(N/4-1)$ follower failures without blocking the fast path. On the other, (C2) has a better complexity, only requiring a majority of replicas to order commands [32,33,36,38], but may block the fast path when any of these replicas fails.

3.2 Dependencies and Key Invariants

As we noted earlier, to ensure linearizability it is enough for replicas to only agree on the order of non-commuting commands. SwiftPaxos represents such an order by associating with each command c the set of its *dependencies*—commands conflicting with c that must be executed before it. To compute this set, the replicas make their *proposals* and then agree on one of them. We say that the command c is *committed* when the replicas agree on all its transitive dependencies, i.e., the dependencies of c, the dependencies of these dependencies, and so on. The protocol ensures:

INVARIANT 1. Any two replicas commit a command with the same set of dependencies.

A command moves through different phases as it is processed, from the initial phase (START) to the final one, where the command is committed (COMMIT). Each replica tracks the progress of commands in an array phase, indexed by command identifiers. The name of the phase written in italics denotes the set of all commands in this phase, e.g., *Commit* stands for $\{id \mid \mathsf{phase}[id] = \text{COMMIT}\}$. An array dep maps a command identifier to its direct dependencies: if the command's entry in the phase array is COMMIT, then the dependencies are committed; otherwise, they record the replica's proposal. The dep array defines the edges of a *dependency graph* at a replica. Initially, all the entries are null ($\perp$).

A replica executes a command once it is committed and all its dependencies are executed. Since dependencies relate only conflicting commands, replicas are thus free to execute independent commands in any order. Then to satisfy the Ordering property of SMR (§2), the protocol needs to ensure:

INVARIANT 2. For any two conflicting commands c and c' committed at a replica, either c belongs to the dependencies of c', or the converse holds.

To illustrate, consider commands $\mathtt{x}$, $\mathtt{y}$ and $\mathtt{z}$ such that $\mathtt{x} \bowtie \mathtt{y}$ and $\mathtt{y} \bowtie \mathtt{z}$ and their committed dependencies are $\mathtt{dep}[\mathtt{id}(\mathtt{x})] = \mathtt{dep}[\mathtt{id}(\mathtt{z})] = \emptyset$ and $\mathtt{dep}[\mathtt{id}(\mathtt{y})] = \{\mathtt{x}, \mathtt{z}\}$. Then a replica can execute $\mathtt{x}$ and $\mathtt{z}$ in any order, provided they execute before $\mathtt{y}$.

Finally, as the protocol delays executing a committed command until its dependencies are executed, for the protocol to be live, committed dependencies should not form a cycle:

INVARIANT 3. The committed part of the dependency graph at each replica is acyclic.

3.3 Agreeing on Dependencies

Before describing SwiftPaxos in detail, we give its overview using the example in Figure 1. Here the system consists of 5 replicas, thus tolerating 2 faults. Replicas p_4 and p_5 host one client each. A third client is not co-located with any of the replicas. The fast quorum intersection requirement (FQI) is satisfied with configuration (C2), so that there is a single fast quorum $\{p_1, p_2, p_3\}$. To avoid clutter we omit most of the interactions with the two replicas outside this quorum.

Propagation. A client sends a submitted command c to the replicas in a `Propagate`(c) message. When a replica receives c, it computes its proposal for c's dependencies as the set of the previously received commands that conflict with c—to satisfy Invariant 2. In Figure 1, proposals are depicted using $\to$, so that $\{\mathtt{x}, \mathtt{y}\} \to \mathtt{z}$ means that $\mathtt{z}$ depends on $\mathtt{x}$ and $\mathtt{y}$.

Fast path. Each replica in a fast quorum broadcasts its proposal in a `FastAck` message. Replicas wait until they receive

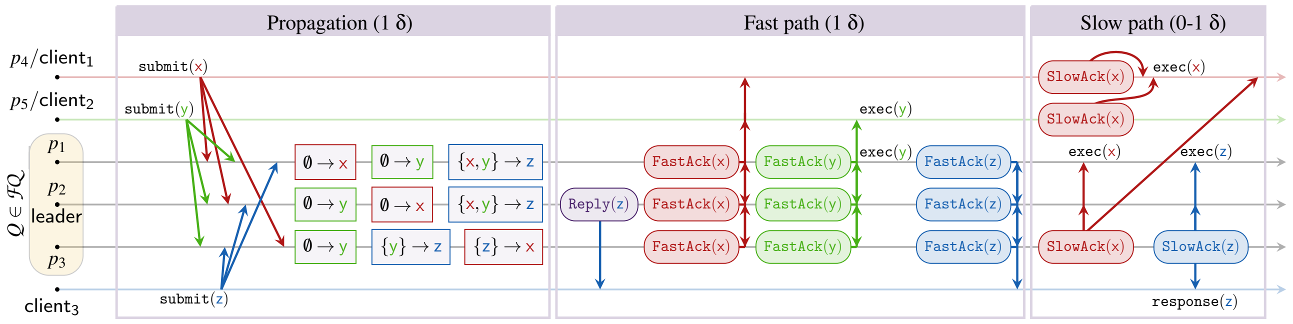


Figure 1: An execution of the protocol processing commands x, y and z such that $x \bowtie z$ and $y \bowtie z$.

proposals from all members of some fast quorum. If all such proposals are the same, then there is a spontaneous agreement [31]. In this case, a replica commits the command once its dependencies are also committed. This represents the *fast path* of the protocol, which under favorable conditions allows a replica to execute a command within 2 message delays from the time it was submitted. For example, in Figure 1 replica p_1 commits and executes y immediately after receiving the same proposal $\emptyset \rightarrow y$ from all fast quorum members. Since each replica computes its proposal from the conflicting commands it has seen before, a command takes the fast path when there is no contention—something that occurs frequently in application workloads [26, 38, 41].

Slow path via double voting in a fast quorum. Fast quorum replicas compute their proposals based on the order in which they receive conflicting commands. As this depends on the ordering generated by the network, the replicas may disagree, in which case the command takes the *slow path*. To this end, each fast quorum replica that disagreed with the leader adopts its proposal and acknowledges this by broadcasting a `SlowAck` message. A replica can commit the command once it receives a set of matching `FastAck` and `SlowAck` messages from all the members of some fast quorum and it has committed the command's dependencies. This may require an extra message delay in comparison to the fast path. For example, in Figure 1 replicas p_1 and p_2 receive x before z, while p_3 receives z before x, which results in different proposals. Since p_3 disagrees with the leader p_2 on the dependencies of x and z, it adopts the leader's proposals $\emptyset \rightarrow x$ and $\{x, y\} \rightarrow z$ and broadcasts `SlowAcks` for the two commands. Consequently, replica p_1 commits and executes x and z in 3 message delays.

Note that on the slow path a fast quorum replica votes for two different proposals in the same ballot—once for its own (`FastAck`) and once for the leader's (`SlowAck`). Such double voting would usually be unsafe in Paxos-like protocols. In SwiftPaxos this is safe because the leader belongs to all fast quorums: if a fast quorum replica disagreed with the leader, the replica knows that the command could not have been committed on the fast path, and may safely correct its vote.

Slow path via a slow quorum. SwiftPaxos can also commit commands using a slow quorum, just like Paxos. A slow quorum replica acts similarly to a fast quorum replica, except that it does not make proposals of its own. Namely, when it receives a `FastAck` from the leader, a slow quorum replica adopts the leader's proposal and broadcasts a `SlowAck` message. Any replica can commit the command after receiving `FastAck` from the leader and `SlowAcks` from the followers of a slow quorum. In a wide-area network, where latencies between different pairs of replicas vary, the slow path may be faster than the double-voting mechanism in the fast path. Going back to Figure 1, assume that the latency between p_4 and p_3 is higher than between p_4 and p_5. Instead of waiting for a `SlowAck` message for x from p_3, replica p_4 can use the earlier `SlowAck` message from p_5 and the `FastAck` from the leader p_2 to commit x via the slow quorum $\{p_4, p_5, p_2\}$. Since a slow quorum can be any majority, this additional mechanism for committing commands ensures that SwiftPaxos is *at least as fast* as Paxos. In fact, on the slow path SwiftPaxos corresponds to a well-known faster variation of Paxos where followers broadcast 2B messages to all replicas.

Message complexity. SwiftPaxos has a quadratic message complexity, as opposed to linear complexity of classical Paxos. However, for typical geo-replicated SMR deployments ($N = 3$ or $N = 5$) [9, 14] the difference between these complexities is fairly small. Besides, the additional messages issued by SwiftPaxos are light, as they only carry metadata. We compare the bandwidth usage of SwiftPaxos and Paxos in §5.1.

3.4 Ensuring Low Tail Latency

Dependencies are also used to order commands in EPaxos [38] and its follow-ups [3, 17]. However, in EPaxos dependencies may form cycles, and thus command execution cannot simply follow their order. Instead, EPaxos waits until it forms strongly connected components of the dependency graph and then executes these components one at a time. Since such components can be arbitrary large, the protocol may delay command execution for an unbounded amount of time. This phenomenon is known as *convoy effect* and in practice it leads to a high tail latency [6, 19, 50].

In contrast, dependencies in SwiftPaxos remain acyclic (Invariant 3, §3.2), and command execution can simply follow

	Dependency graph	**Dependencies of c_3**	**Paths to c_3**
leader	$c_1 \to c_2 \to c_3$	$\{c_2\}$	$\{[c_3, c_2, c_1]\}$
p_1	$c_2 \to c_3$	$\{c_2\}$	$\{[c_3, c_2]\}$
p_2	$c_2 \to c_3$	$\{c_2\}$	$\{[c_3, c_2]\}$

Figure 2: To accept the result of optimistic execution, clients wait for matching dependency paths.

them. The only delay between committing a command and executing it is similar to one present in Multi-Paxos, where each consensus instance depends on the prior ones. This makes the tail latencies of SwiftPaxos and Paxos comparable, as we empirically show in §5.2. We compute the theoretical latency of SwiftPaxos and compare it against other protocols in §A.6.

3.5 Faster Responses at Non-Collocated Clients

Clients located in the same data center as a replica receive the result of a command directly from that replica. For instance, client_2 in Figure 1 is co-located with p_5 and receives the response to y immediately after p_5 executes y. Clients that are not co-located with a replica need to wait for an extra message delay to hear the response. To speed up delivering responses at such clients, SwiftPaxos optimistically executes commands at the leader (similarly to Zookeeper [23] and CURP [41]).

In more detail, when the leader receives a command c from a client that is not co-located with a replica, it computes the result of executing c and replies to the client with a `Reply` message. In Figure 1, client_3 gets this message for command z. Followers send their `FastAck` and `SlowAck` messages for c not only to other replicas—as described before—but also to the client that submitted c. The client accepts the response in `Reply` once it receives matching `FastAck` or `SlowAck` messages from a fast or slow quorum. This ensures that a client always gets a response within 2 or 3 message delays, regardless of whether it is co-located with a replica.

There is a subtlety, though: on the fast path a client cannot accept the result of optimistic execution using only the dependency sets, because these sets contain direct dependencies of the command but say nothing about their predecessors. Figure 2 illustrates this. It depicts the dependency graphs at some point in time where (FQI) from §3.1 is satisfied with a single quorum $\{p_0, p_1, p_2\}$, with p_0 as the leader. In Figure 2, clients submit 3 commands c_1, c_2 and c_3, with c_2 conflicting with both c_1 and c_3. The replica p_0 is the only one to receive c_1, yet all three replicas agree on the ordering of c_2 and c_3. Assume that p_0 now optimistically executes c_1, c_2 and c_3 in this order before it gets any messages from other replicas, and thus before this ordering is durable. It will then send the result of executing c_3 to the client. The leader's proposal for the dependency set of c_3 matches the proposals of p_1 and p_2, but the three replicas disagree on c_3's transitive dependencies. Hence, the client cannot accept the reply from the leader: if the leader crashes, its ordering of c_1 will be lost, invalidating the optimistic execution. Notice that this problem does not occur at replicas, as they execute each command only after it gets committed, implying that all its transitive dependencies are also committed, and thus durable.

To ensure that clients accept only valid results from optimistic execution, `FastAck` messages contain not only the direct dependencies of the command but also the set of all the paths leading to it in the dependency graph. If these sets match, the ordering of the command is durable and the client may accept the result via the fast path. For a command c, we represent the set of c's dependency paths as a set of ordered lists, where each list starts with c and continues with its transitive dependencies arranged according to the dependency graph (see Figure 2). In §4.1 we explain how dependency paths can be efficiently implemented in practice.

In Figure 1 when the fast quorum replicas receive z, their `FastAck` replies carry not just dependency sets, but also dependency paths: replicas p_1 and p_2 send $\{[z, x]; [z, y]\}$, while p_3 sends $\{[z, y]\}$. Since these sets do not match, the result of z cannot be accepted by client_3 on the fast path.

4 SwiftPaxos in Detail

We define the protocol logic in Figures 3–5 using a set of handlers, each of which executes atomically once its preconditions are true (keyword **pre**).

4.1 Normal Operation

Propagation. Upon receiving a `Propagate`(c) message from a client (lines 2 and 6), a replica p saves c in an array cmd and moves c to the PREACCEPT phase, to record that it is now waiting for the leader's proposal. If the replica belongs to some fast quorum, it also computes its proposal `dep`$[$`id`$(c)]$ for c's dependencies and broadcasts it in a `FastAck` message together with c's dependency paths `paths`$[id]$ (line 21).

If p is the leader, upon receiving a command p also executes it and sends its result in a `Reply` message (lines 16-19). The command is not committed at this point. The execution is optimistic and p does not modify its local copy of the state machine. Instead, p maintains a list pending_log of commands that are received but not committed yet. The command is added to the list and p determines its result using a function `opt_exec` (further discussed in §4.4).

Fast path. A replica p commits a command c via the fast path when it receives matching dependency sets from a fast quorum. In this case, p first processes the `FastAck` message from the leader (line 22), which advances c to the ACCEPT phase (line 24). Then p processes `FastAck`s from the other quorum members (line 30). Once the dependencies of c are committed (line 31), p commits c as well. A client receives a response on the fast path once it gets a `Reply` message from the leader and matching dependency paths from the followers of a fast quorum (line 3).

```
1  function submit($c$):
2      send Propagate($c$) to $\mathcal{R}$
3  when received Reply($b, id, P, r$) from leader($b$) and
                 FastAck($b, id, \_, P$) or SlowAck($b, id$)
                 from all followers in $Q \in \mathcal{FQ}(b)$, or
4  when received Reply($b, id, \_, r$) from leader($b$) and
                 SlowAck($b, id$) from all followers in
                 $Q \in \mathcal{SQ}(b)$
5      response($id, r$)
```

Figure 3: Client code.

Slow path. Upon the receipt of FastAck(b,id,D,P) from the leader, a fast quorum replica checks whether its local value of dep$[id]$ is equal to D (line 25). If this is not the case, the command takes the slow path: the replica overwrites dep$[id]$ with D, advances the command's phase to ACCEPT and notifies the other replicas and the client with a SlowAck message. A slow quorum replica also sends the SlowAck to speed up agreement on dependencies, as described in §3.3. To commit the command, the replica then waits until the other fast (or slow) quorum members send either a FastAck matching the leader's proposal or a SlowAck (line 30).

The client acts similarly to compute the response to its command, except that it waits for matching dependency paths instead of dependency sets (line 3). This justifies line 29: when a fast quorum replica receives the leader's proposal that matches its own vote but does not match the dependency paths, it sends a SlowAck message to the client. In this way the client learns that the replica is now in sync with the leader and can thus accept the result of the optimistic execution.

To ensure that the replica is indeed in sync with the leader, we need the last conjunct of the precondition at line 23. This requires the replica to handle a FastAck$(_,_,D,_)$ message from the leader only if it has already handled such messages from all the commands in D. This condition is automatically satisfied when communication channels are FIFO. To illustrate its role, imagine that in Figure 1 replica p_3 updated the dependencies of z before those of x, violating the precondition in line 23. Then right after p_3 changed dep[z] to $\{$x$,$y$\}$, the three replicas of the fast quorum disagree on the set of dependency paths of z: at replicas p_1 and p_2 the set is $\{[$z$,$x$];[$z$,$y$]\}$ but according to p_3 there is a path $[$z$,$x$,$z$]$. As seen in §3.5 accepting the result of the execution of z would be unsafe.

Command execution. A replica keeps track of commands executed on its copy of the state machine using a variable Exec. A replica executes a command once it is committed and all its dependencies are executed (line 33). If the replica is the leader, it then removes the command from pending_log.

Representing dependency paths. Sending full dependency paths is not an option in practice due to their fast-growing size. This issue is solved by trimming the paths of a command c up to the last committed or accepted transitive dependency. For example, if c's set of dependency paths is $\{[c,d,e]\}$ and

```
6  when received Propagate($c$) from client $q$
7      pre: status $=$ NORMAL $\wedge$ id($c$) $\in$ $Start$
8      phase[id($c$)] $\leftarrow$ PREACCEPT
9      cmd[id($c$)] $\leftarrow c$
10     if $fast(p,$ bal$)$ then
11         let $D = \{id \mid$ cmd$[id] \bowtie c\}$
12         let $P =$ pset(id($c$))
13         dep[id($c$)] $\leftarrow D$
14         paths[id($c$)] $\leftarrow P$
15         if $p =$ leader(bal) then
16             pending_log $\leftarrow$ pending_log $\cdot\, c$
17             let $r =$ opt_exec(pending_log, state)
18             send Reply(bal, id($c$), $P$, $r$) to $q$
19             send FastAck(bal, id($c$), $D$, $P$) to $\mathcal{R}$
20         else
21             send FastAck(bal, id($c$), $D$, $P$) to $\mathcal{R} \cup \{q\}$

22 when received FastAck($b, id, D, P$) from leader($b$)
23     pre: status $=$ NORMAL $\wedge$ $id \in Preaccept$ $\wedge$ bal $= b$
               $\wedge\, D \subseteq Accept \cup Commit$
24     phase[$id$] $\leftarrow$ ACCEPT
25     if $(fast(p,b) \wedge$ dep$[id] \neq D) \vee slow(p,b)$ then
26         dep[$id$] $\leftarrow D$
27         send SlowAck(bal, $id$) to $\mathcal{R} \cup \{$client($id$)$\}$
28     else if $fast(p,b) \wedge$ paths$[id] \neq P$ then
29         send SlowAck(bal, $id$) to client($id$)

30 when received FastAck($b, id, D, \_$) or SlowAck($b, id$)
       from all followers in $Q_f \in \mathcal{FQ}(b)$, or
       SlowAck($b, id$) from all followers in $Q_s \in \mathcal{SQ}(b)$
31     pre: status $=$ NORMAL $\wedge$ $id \in Accept$ $\wedge$ bal $= b$
               $\wedge\, D =$ dep$[id] \subseteq Commit$
32     phase[$id$] $\leftarrow$ COMMIT

33 when there exists $id \in Commit \setminus$ Exec with dep$[id] \subseteq$ Exec
34     ($\_$, state) $\leftarrow$ exec(cmd[$id$], state)
35     Exec $\leftarrow \{id\} \cup$ Exec
36     if $p =$ leader(bal) then remove(pending_log, $id$)

37 function pset($id$):
38     if dep$[id] = \emptyset$ then return $\{[id]\}$
39     else return $\{id :: l \mid \exists id' \in$ dep$[id].\, l \in$ pset$(id')\}$
```

Figure 4: Normal operation at a replica p.

d is in the COMMIT or ACCEPT phase, the set is reduced to $\{[c,d]\}$. Thanks to the preconditions at lines 23 and 31, this is safe: if a command is committed or accepted then so is any of its transitive dependencies. When the reduced sets match, the omitted parts match as well, as they correspond to the ordering at the leader. Additionally, instead of the paths themselves one may send hashes of the (reduced) paths [44]. This optimization is also applied in our implementation.

Garbage collection. Standard periodic checkpoints can be used to bound memory usage of the protocol [7,23]. In our implementation, we simply trim a command from the protocol's state once it is executed at all replicas. In §5.1 we discuss the memory consumption in more detail.

```
40 function recover():
41     let $b > \mathsf{bal}$ such that $\mathsf{leader}(b) = p$
42     send NewLeader($b$) to all

43 when received NewLeader($b$) from $q$
44     pre: $b > \mathsf{bal}$
45     $\mathsf{status} \leftarrow$ RECOVERING
46     $\mathsf{bal} \leftarrow b$
47     send NewLeaderAck($b$, cbal, phase, cmd, dep) to $q$

48 when received NewLeaderAck($b, cbal_q, phase_q, cmd_q, dep_q$)
       from all $q \in Q$
49     pre: $\mathsf{status} =$ RECOVERING $\wedge\ \mathsf{bal} = b \wedge |Q| > n/2$
50     reset phase, cmd and dep
51     let $b_{\max} = \max\{cbal_q \mid q \in Q\}$
52     let $U = \{q \in Q \mid cbal_q = b_{\max}\}$
53     forall $id$ such that
         $\exists q \in U.\, phase_q[id] \in \{$ACCEPT, COMMIT$\}$ do
54         $\mathsf{phase}[id] \leftarrow phase_q[id]$
55         $\mathsf{cmd}[id] \leftarrow cmd_q[id]$
56         $\mathsf{dep}[id] \leftarrow dep_q[id]$
57     forall $id \notin Accept \cup Commit$ do
58         if $\exists D \neq \bot.\, \exists Q_f \in \mathcal{FQ}(b_{\max}).\, Q \cap Q_f \subseteq U$
             $\wedge\, \forall q \in Q \cap Q_f.\, dep_q[id] = D$ then
59             $\mathsf{phase}[id] \leftarrow$ ACCEPT
60             $\mathsf{cmd}[id] \leftarrow cmd_q[id]$
61             $\mathsf{dep}[id] \leftarrow$
                 $D \cup \{id' \mid \mathsf{cmd}[id'] \bowtie \mathsf{cmd}[id] \wedge id \notin \mathsf{dep}[id']\}$
62     forall $(id, id')$ such that $id' \in Start \cap \mathsf{dep}[id]$ do
63         $\mathsf{dep}[id] \leftarrow \mathsf{dep}[id] \setminus \{id'\}$
64     arbitrarily break cycles in dep
65     send Sync($b$, phase, cmd, dep) to $\mathcal{R}$

66 when received Sync($b, phase, cmd, dep$)
67     pre: $b \geq \mathsf{bal}$
68     $\mathsf{status} \leftarrow$ NORMAL; $\mathsf{bal} \leftarrow b$; $\mathsf{cbal} \leftarrow b$
69     $\mathsf{phase} \leftarrow phase$; $\mathsf{cmd} \leftarrow cmd$; $\mathsf{dep} \leftarrow dep$
70     clear(pending_log)
71     if $slow(p, b)$ then
72         forall $id$ in an order consistent with dep do
73             if $p = \mathsf{leader}(b) \wedge id \notin \mathsf{Exec}$ then
74                 $\mathsf{pending\_log} \leftarrow \mathsf{pending\_log} \cdot \mathsf{cmd}[id]$
75             else if $id \in Accept$ then
76                 send SlowAck($b, id$) to $\mathcal{R}$
```

Figure 5: Recovery at a replica p.

4.2 Recovery from Leader Failures

Replicas continuously monitor the progress of the protocol. When the current leader is suspected of hindering progress, the protocol nominates a new one to take over. This nomination can be done in a standard way, e.g., using a failure detector [8], and we defer the details to §A.4. We now explain the algorithm followed by the potential leader.

Leadership change. The recovery procedure begins similarly to Paxos [29]. When a replica p is nominated to become the new leader, it calls `recover` (line 40). This function picks a new ballot b led by p and higher than any ballot p has previously joined. Replica p then sends b in a `NewLeader` message to all replicas, asking them to support its leadership. A replica acknowledges p as the new leader only if b is higher than any ballot it has previously joined (line 44). In this case the replica changes its status to RECOVERING, which stops normal message handling, and replies with a `NewLeaderAck` message carrying the commands it knows about.

Once the leadership of p is approved by a quorum Q (line 48), p's next goal is to bring the replicas into the same state from which they will resume processing commands.

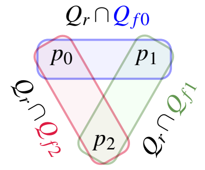
Recovering commands. To maintain consensus on dependencies (Invariant 1), the new leader's state must include all commands committed in lower ballots. To ensure this, the leader computes the initial state of b based on the states and the values of a cbal variable reported by the replicas in their `NewLeaderAck` messages. This variable maintains the last ballot at which each replica successfully completed recovery.

Similarly to Paxos [29], p focuses on the set U of replicas that reported the maximal ballot $b_{\max}$: the state of these replicas supersedes that of replicas from lower ballots. The leader p then incorporates into its state all the commands that could have been committed up to $b_{\max}$. In lines 53–56 the leader does this for commands that could have been committed on the slow path. Any such command would have to be accepted by a slow quorum Q_s, which must intersect with the recovery quorum Q in at least one process. Hence, for each command c in the ACCEPT or COMMIT phase at a replica $q \in U$, the leader incorporates c and its dependencies as reported by q into its state. In lines 58–61 the leader collects all commands that could have been committed up to $b_{\max}$ on the fast path. Any such command would have to be preaccepted with the same dependencies by a fast quorum Q_f. Hence, p adds to its state all commands c such that $Q \cap Q_f \subseteq U$ and all replicas in $Q \cap Q_f$ report the same dependency set D for c. This condition is similar to Fast Paxos [31]; we later explain the rationale for the second term of the union at line 61 and how we ensure that we don't create cycles in the dependency graph. Note that the condition at line 58 defines D uniquely: there may not be two different fast quorums Q_f and Q'_f whose members in Q report different dependency sets. This is because (FQI) implies $(Q \cap Q_f) \cap (Q \cap Q'_f) = Q \cap Q_f \cap Q'_f \neq \emptyset$. But then the dependencies reported for the command c by $Q \cap Q_f$ and $Q \cap Q'_f$ cannot be distinct, since they were reported by some replica belonging to both sets.

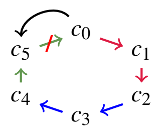
Thanks to the conditions at lines 53 and 58, all commands committed at earlier ballots are included into the new state. Next, the leader removes from the dependency graph all commands that do not satisfy those conditions (lines 62-63). To see that the loop at line 62 does not violate safety properties, note that for a pair of commands c and c' whose identifiers id and id' satisfy the condition at line 62, neither command can be committed at a previous ballot: c' is not committed because it does not satisfy any of the two conditions at line 53 and line 58; c is not committed because it depends on an uncommitted command c'. Thus, it is safe to completely remove c' from the state and update the dependencies of c.

$$\begin{array}{ll} Q_{f0} = \{p_0, p_1, p_3, p_4\} & p_0 :\ c_0 \to c_1 \to c_2 \to c_3 \to c_4 \\ Q_{f1} = \{p_1, p_2, p_3, p_4\} & p_1 :\ c_2 \to c_3 \to c_4 \to c_5 \to c_0 \\ Q_{f2} = \{p_0, p_2, p_3, p_4\} & p_2 :\ c_4 \to c_5 \to c_0 \to c_1 \to c_2 \\ Q_r = \{p_0, p_1, p_2\} & F = \{p_3, p_4\} \end{array}$$

(a)



(b)



(c)

Figure 6: Avoiding dependency cycles during recovery.

After the above computation, the leader breaks cycles in the dependency graph; we explain this step shortly. The leader then broadcasts a `Sync` message, defining the state from which the new ballot starts. When another replica receives this message (line 66), it overwrites its state with the one provided, changes its status to NORMAL, and clears the log of pending commands. If the replica belongs to a slow quorum, it broadcasts a `SlowAck` message for each uncommitted command (line 76), to ensure that such commands are committed in the new ballot. Finally, if the replica is the new leader, it adds all unexecuted commands to `pending_log` in an order consistent with the dependencies (line 72).

Recovery can be easily optimized so that `Sync` messages only transfer the parts each replica is missing (although our prototype implementation omits this optimization).

Preserving the invariants. During the normal protocol operation, both on the fast and slow paths replicas commit commands with the dependencies proposed by the leader. Then, due to the way the leader computes its proposals, committed conflicting commands cannot be independent, and the dependencies of committed commands cannot form a cycle, as respectively required by Invariants 2 and 3. However, preserving these invariants during recovery requires care.

First, during recovery the new leader can introduce a cycle when it merges replica states. To see how this can happen, consider the example illustrated in Figure 6. The system consists of five replicas and recovers from $Q_r = \{p_0, p_1, p_2\}$ after the failure of the remaining replicas $F = \{p_3, p_4\}$. Assume that on recovery all replicas in Q_r report the same $\mathsf{cbal} = b$ and the orderings shown in Figure 6a. Assume also that the leader of b is p_4, and that this ballot is associated with three fast quorums Q_{f0}, Q_{f1} and Q_{f3}, defined in the figure. On recovery the replicas in $Q_r \cap Q_{f0}$, $Q_r \cap Q_{f1}$ and $Q_r \cap Q_{f2}$ agree on the orders $c_2 \to c_3 \to c_4$, $c_4 \to c_5 \to c_0$ and $c_0 \to c_1 \to c_2$, respectively (Figure 6b). According to line 58, the new leader has to incorporate all of them into its state. But when these orderings are combined, they yield a cycle over commands $c_0, \ldots, c_5$. To preserve Invariant 3 in such cases, the leader arbitrarily breaks cycles by inverting relations between any two commands in them (line 64). For example, the cycle in Figure 6 can be broken by removing c_5 from $\mathsf{dep}[\mathtt{id}(c_0)]$ and adding c_0 to $\mathsf{dep}[\mathtt{id}(c_5)]$ (Figure 6c). Note that this computation does not violate Invariant 1 because it changes only those dependencies that could not have been committed in previous ballots. Namely, if *some* command on the cycle is committed, then so are all its predecessors, i.e., *all* commands on the cycle (by line 31); but this is impossible by Invariant 3. We refer to §A for a detailed proof.

When computing the new state during recovery, the new leader can also end up with two conflicting commands none of which is a dependency of the other. To see how this can happen, imagine that in Figure 1 recovery occurs right after the moment p_3 received $\mathtt{FastAck}(b, \mathsf{x}, \emptyset, \_)$ from the leader. At this moment the replica has x in the ACCEPT phase with an empty set of dependencies, and z in the PREACCEPT phase, with y as its only dependency. Incorporating both commands into the new state with the dependencies reported by p_3 would violate Invariant 2, because neither of the conflicting commands x and z is a dependency of the other. To avoid such situations, the new leader updates dependency sets of the commands that are not in the ACCEPT or COMMIT phase at some replica of the recovery quorum (line 61). The leader adds to the dependencies of c all conflicting commands, excluding those that already depend on c. In the above example, the leader includes x in the dependency set of z (line 61), which is safe because z is not committed at previous ballots. In §A we prove that this computation is also safe in general.

4.3 Recovery from Client and Follower Failures

Before voting on a command, a replica must know its payload (lines 6 and 23), which may not be available locally if the client has failed. To deal with this, if the replica receives an acknowledgment from the leader regarding this command, after some time it tries to fetch the payload from the leader. Additionally, followers re-propose commands if they are not committed after some timeout. This helps when the payload is available at followers but missing at the leader.

Follower failures do not impact the availability of SwiftPaxos. But the failure of a follower within the fast quorum can disable the fast path of the protocol and degrade its performance. SwiftPaxos can deal with this using the same procedure as for leader failures, changing the ballot to one with a different fast quorum.

4.4 Optimistic Execution

The execution performed by the leader to compute the result of a command (line 17 in Figure 4) is optimistic: at the time the leader invokes `opt_exec`, the command has not yet been accepted by the followers. However, the followers will always accept the command unless one of them suspects the leader of failure, which happens rarely. Hence, the work done during the optimistic execution is rarely lost.

The optimistic execution in SwiftPaxos can be implemented using existing mechanisms [25, 28, 41, 49]. For example, the command's response can be computed by reading the local copy while taking into account prior uncommitted commands in pending_log, and its side effects can be buffered until commit. Such a mechanism is already at work in industrial-grade systems, such as Zookeeper [23].

SwiftPaxos can also be optimized by speculatively executing read-only commands not at the leader, but at any fast quorum replica. A client receives a reply from this replica and, as before, accepts it on the fast path when other fast quorum members report matching dependency paths. This optimization distributes the load across replicas, preventing the leader from saturating. We assess its benefits in §5.6.

5 Evaluation

Our evaluation compares SwiftPaxos against several other protocols, as detailed next. All protocols are written in Go, building on the codebase of EPaxos [38]. The source code is publicly available [1].

***Paxos* [29].** Commands get ordered and disseminated by the leader replica, which is also in charge of replying to clients.

***N^2Paxos*.** A variation of Paxos that broadcasts 2B messages (corresponding to our `SlowAcks`) to all replicas. A client receives the response from the site closest to it. This cuts one message delay for clients close to a replica, allowing them to learn a response in 3 message delays instead of 4.

***FastPaxos+* [31].** Clients send their requests directly to all replicas, and a command is committed if enough replicas agree on its ordering. When replicas disagree, the protocol starts a new ballot of N^2Paxos. We implement uncoordinated collision recovery [31] (hence the +), which reduces commit latency in exchange for a fixed collision-recovery quorum. In this version replicas locally compute a proposal for the next ballot, bypassing the coordinator and saving one round trip.

***GPaxos* [30].** Generalized Paxos improves Fast Paxos by increasing fast path rate using the commutativity of commands. However, it needs heavy metadata that requires frequent checkpointing [20, 47].

***CURP+*.** CURP [41] boosts leader-based SMR protocols by separating ordering from durability. Each command is optimistically executed at the leader while being stored at $> 3/4$ of replicas ("witnesses"). If there is no conflict, the result of the optimistic execution at the leader is usable right away, without waiting for the command to commit. CURP is a primary-backup replication protocol, i.e., it uses only $f+1$ replicas but requires accurate failure detection. We use its variant for $2f+1$ replicas [41, Appendix B.2] based on N^2Paxos, to reduce latency (hence the +). See §B for the details.

***EPaxos* [38].** A leaderless protocol where clients connect to the closest replica, which coordinates access to the replicated service. In the conflict-free case, clients co-located with a replica know the response after 2 message delays. Far-away clients need an extra round trip to receive it. We deploy EPaxos in thrifty mode, which reduces the overall number of sent messages and the size of fast quorums.

***Mencius* [37].** Mencius rotates the role of the consensus leader among the replicas. This spreads the load, but makes the system run at the speed of the slowest replica.

We use two benchmarks: a no-op service and a key-value store with an API following the one of YCSB [13]. The data model is a set of records that are accessed using the commands `insert`, `get`, and `update`. Each record is stored under some key. Two commands conflict when they access the same record and one of them is a write. The no-op service executes commands accessing a random key and carrying a default payload of 1 KB (the standard YCSB payload size). Two commands conflict when they are on the same key. Following standard practice [38], to measure performance under a conflict rate ρ a client chooses key 0 with a probability ρ, and some unique key otherwise.

We deploy the services on Amazon EC2 over 5 replicas in different geographical regions, so that the system tolerates both a failure and a planned outage due to maintenance—a common deployment configuration [14]. Clients are spread over 10 regions all around the world, 2 of which also host replicas. Hence, our experiments use 13 EC2 regions in total (see §C for more details). Both clients and replicas execute in virtual machines running Amazon Linux 2 with 16 vCPUs and 32 GB of main memory. Unless stated otherwise, clients execute commands in a closed loop, waiting for the previous command to return before submitting a new one. Leader-based protocols execute with the leader placed at the site that minimizes the average (mean) latency across all clients. By default, our experiments use configuration (C2) to satisfy (FQI) for both SwiftPaxos and FastPaxos+, as it is slightly more favorable than (C1). We investigate the trade-offs between the two configurations in §5.5.

5.1 Impact of the Conflict Rate

Our first experiment varies the conflict rate from 0 to 100% in 10% increments. Each site hosts 100 clients (1000 clients in total). Figure 7*(a)* presents the speedup relative to Paxos. The latency experienced by clients under each protocol depends on where they are relative to the replicas. The top row of Figure 7*(a)* reports the average (mean) latency across all sites. The middle row reports the speedup observed at the best site for SwiftPaxos, and the bottom row reports the worst.

The latencies of Paxos, N^2Paxos, and FastPaxos+ are independent of the conflict rate. On average the improvement of N^2Paxos over Paxos is slightly below 1.05x: from 239 ms to 229 ms. This is due to clients co-located with replicas, which benefit from the broadcast of 2B messages. FastPaxos+ has an almost null fast path ratio because it is rare that the replicas receive two concurrent commands (conflicting or not) in the same order. Its slow path follows the message

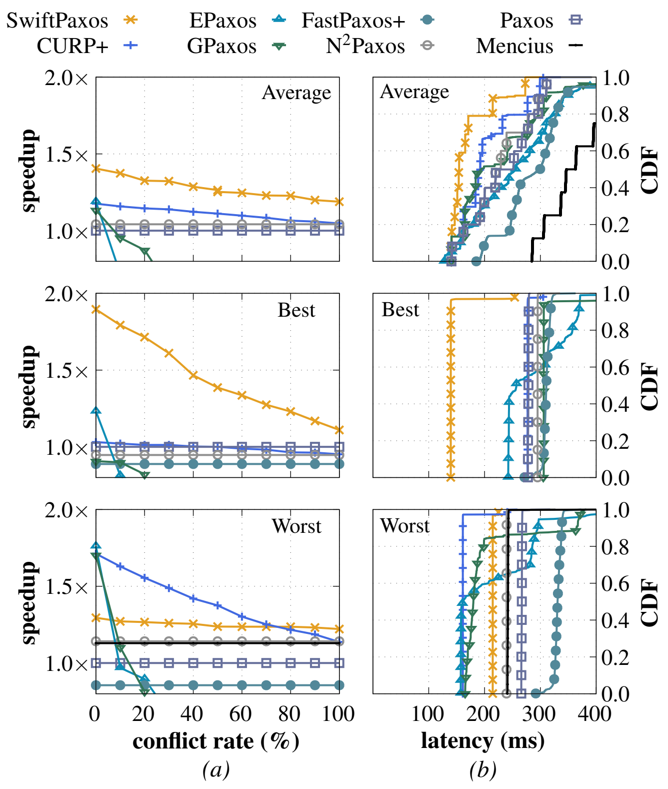


Figure 7: *(a)* Speedup over Paxos when varying the conflict rate and *(b)* latency distribution under 2% conflicts. From top to bottom, we report the average (mean), best, and worst sites for SwiftPaxos.

flow of N^{2}Paxos with one difference: to take the slow path, a replica must first detect a collision (with a majority of 2B messages). This explains why FastPaxos+ is slower than N^{2}Paxos: 300 ms against 229 ms. For this reason, the average latency of FastPaxos+ is omitted from Figure 7*(a)*. The same happens with Mencius, which displays a latency of 360 ms on average.

The average latency of SwiftPaxos is between 170 ms and 201 ms. With a 0% conflict rate this is 1.4x faster than Paxos. Each 10% increase in the conflict rate yields a 1–3% increase in latency. With 100% of conflicts, the speedup is 1.19x. At 7 out of 10 client sites SwiftPaxos is more than 20% faster than Paxos, achieving more than 40% improvement at 3 of them.

At a low conflict rate, EPaxos and GPaxos are faster than Paxos, yet both see their performance decrease abruptly as conflicts increase. This is mainly explained by the drop in the fast path ratio. Another reason is that at 8 out of 10 client sites there is no service replica. These clients need an extra round-trip to get a reply. In EPaxos, performance also deteriorates due to the convoy effect in command execution (§3.4). Thanks to its optimistic execution mechanism, CURP+ is the second best protocol, with a 1.18x speedup on average. Similarly to other protocols, its performance decreases with conflicts.

SwiftPaxos is faster than other protocols at all sites except one (last row in Figure 7*(a)*): this site is too far away from the fast quorum to leverage it. The improvement over CURP+

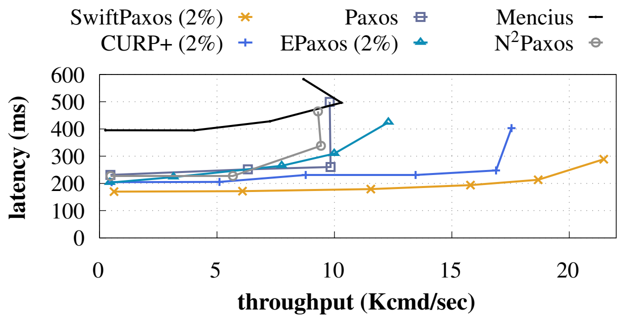


Figure 8: Saturation points for 5 replicas when the total number of clients increases from 100 up to 5000.

comes from a better fast path condition and smaller fast path quorums.

5.2 Tail Latency

Figure 7*(b)* reports the cumulative distribution function (CDF) of the latency at clients with 2% of conflicts. As before, we run 100 clients per site. EPaxos has a suboptimal tail latency distribution due to a convoy effect in its execution mechanism (§3.4). GPaxos has similar issues because it requires complex data structures and needs to regularly start new ballots to minimize metadata. Other leader-based protocols do not suffer from tail latency issues.

5.3 Metadata Usage

The leader in Paxos sends the payload of each command to the quorum of replicas. This is not the case with SwiftPaxos, because the protocol uses command identifiers for ordering. As a consequence, SwiftPaxos can be cheaper than Paxos when contention is rare. With 0% conflicts, metadata in SwiftPaxos consumes 2.83 GB in the experiment of Figure 7*(a)*. This is 1.16x better than Paxos (3.27 GB). The protocol message complexity increases with more conflicts. In Figure 7*(a)* with 100% conflicts data consumption is 3.36 GB, around 3% more than Paxos. At a replica, the dependency graph grows over time, but shrinks due to garbage collection (see §4.1). In Figure 7*(a)*, the average size of a command's dependency set is only 24 B. On average, the dependency graph at a replica is about 8% of the protocol's memory usage.

5.4 Scalability

To evaluate the scalability of SwiftPaxos, we run an experiment that progressively brings it to saturation. The results are reported in Figure 8. The total number of clients increases from 100 to 5000, and the payload of each command is set to 3 KB (this payload size is chosen so as to saturate the system with a reasonable number of client machines).

During this experiment, Paxos and N^{2}Paxos display similar behavior. The two protocols saturate when 2500 clients are deployed across the 10 sites. This is due to the fact that the leader is in charge of broadcasting the commands to the

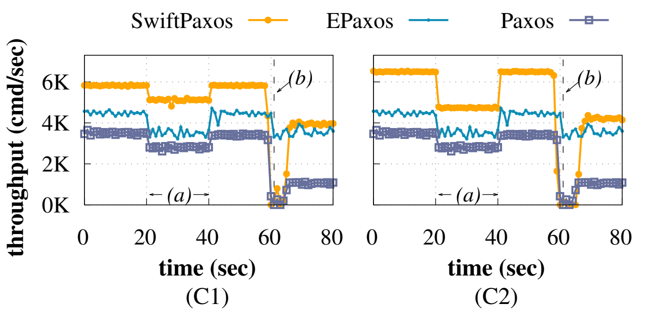


Figure 9: An execution for configurations (C1) and (C2) where *(a)* a replica slows down and *(b)* the leader fails.

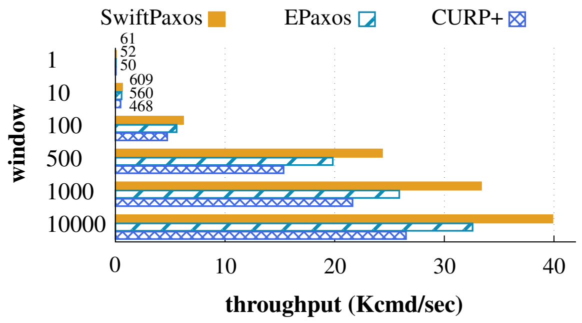


Figure 10: Throughput under a pipelined workload with different window sizes.

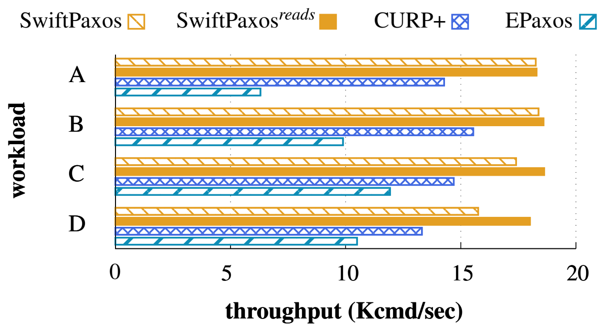


Figure 11: Throughput under different YCSB workloads.

replicas. In the absence of conflicts, the throughput of EPaxos is around 24% higher than that of Paxos (7.8K ops/s against 6.3K ops/s) at almost the same average latency ($\approx$260 ms). When we further increase the system load, responsiveness is affected. Saturation occurs at 400 clients per site, where the system delivers 12K ops/s at an average latency of 424 ms.

In contrast to Paxos, leader-based algorithms with fast paths (CURP+ and SwiftPaxos) do not funnel commands through the leader. Instead, each client is responsible for sending its commands to the replicas. This increases the throughput of the protocol by (at least) 30% on average.

5.5 Performance under Asynchrony

Figure 9 compares how SwiftPaxos, EPaxos, and Paxos behave under asynchrony. We run this experiment for both configurations (C1) and (C2) with 100 clients per site. For (C2) the fast quorum of SwiftPaxos and the lowest-latency quorum of Paxos are the same. For both configurations after 20 s, the latency at one site increases by 200 ms, and it drops back to normal after another 20 s (interval *(a)*). Then the leader fails (event *(b)*).

Before and after the first slowdown, SwiftPaxos performs slightly better in configuration (C2), confirming that a single majority fast quorum is beneficial in our configuration. However, during the slowdown the throughput of (C2) decreases by 27%, whereas it decreases only by 12% for (C1). The degradation is smoother in the latter case because (C1) allows multiple fast quorums, and thus makes the fast path more robust. In Figure 9, SwiftPaxos always outperforms Paxos. This is because, in the worst case, the protocol can commit a command using a slow quorum (see §3.3). Upon event *(b)*, the two algorithms recover at similar speeds. The gap is the largest with configuration (C2) where SwiftPaxos and Paxos take respectively 7 s and 6 s. EPaxos is the most stable protocol because it is leaderless: during the two experiments, its throughput never drops by more than 20%.

5.6 Applications

We now evaluate the protocols under two representative application scenarios. In a first scenario, we consider applications that are pipelining state-machine commands. Such a pattern occurs in log replication as well as in pub/sub architectures. In the second scenario, we run the Yahoo! Cloud Serving Benchmark (YCSB) [13].

Pipelining (Figure 10). In this synthetic benchmark clients pipeline commands to the protocol. This pattern is common in distributed applications, for instance in pub/sub systems such as Apache Kafka [27]: in this case all the messages published by the producer are appended to a log before being read by the consumers. A similar pattern is used when updating an object o accessible through a reference r [23]. The client creates a new version o' of o via asynchronous changes, and then atomically updates r to store the address of o'.

In this experiment, we deploy a client at every datacenter. Each client pipelines commands accessing the same key. Each command carries a payload of 4 KB. The window parameter determines the maximum number of in-flight commands.

Even though commands are received in the same order at the replicas, CURP+ never takes the fast path. This is because witnesses in CURP+ cannot process more than one conflicting command at a time. Conversely, because the run is contention-free, replicas in EPaxos and SwiftPaxos compute the same dependencies, enabling the fast path. On average, both protocols improve over CURP+ by 24% and 49%, respectively.

YCSB (Figure 11). This benchmark consists of workloads with zipfian-like access patterns: workload A is update-heavy (20% `update`, 80% `get`), B is read-heavy (95% `get`, 5% `update`) and C is read-only. Workload D contains repeated reads (95% of the calls) mixed with insertions of new records. The key-value store service holds 10^6 records. A record has 10 fields of 800 B each (8 KB in total). We run 4000 YCSB clients scattered around the world, 400 per site.

In Figure 11, SwiftPaxos^{reads} indicates that read requests are optimistically executed at the closest fast quorum replica, and thus not necessarily at the leader (§4.4). Paxos is not evaluated because, as shown in Figures 7 and 8, EPaxos performs better than Paxos under low conflict rates. This is the case in YCSB, as the benchmark is read-dominated.

EPaxos is most efficient with the read-only workload (C). In this case, all commands execute after a single round trip to the closest quorum, and only the replica that submitted a command executes it. In Workload C, the performance of CURP+ and SwiftPaxos without optimized reads is limited, as only the leader responds to clients. When reads are optimistically executed at any fast quorum replica, SwiftPaxos^{reads} improves over the base version of the protocol, from 17K ops/s to 19K ops/s. In Workload D, the protocols behave similarly to Workload C, as there are also no conflicts.

Workloads A and B contain respectively 20% and 5% of writes. The access distribution is zipfian, with 20% of chance to access the 12 most popular records. In Workload B, CURP+ and SwiftPaxos are around 1.8x faster than EPaxos. This difference is due to the high ratio of slow paths and the convoy effect: 5% of the slowest commands take 470 ms to execute. This performance gap further increases with workload A, where CURP+ and SwiftPaxos provide respectively 2.2x and 2.9x improvement over EPaxos.

6 Related Work

Standard SMR protocols [22, 29, 39] are leader-driven, requiring 4 message delays for a client to receive a response: a round-trip from the client to the leader plus a round-trip from the leader to the replicas. Replicas can compute the response earlier if they exchange Paxos 2B messages. This cuts one message delay for clients close to a replica, but faraway clients still need one more message delay to get notified. We evaluate this variation of Paxos in §5. SDPaxos [52] separates durability from ordering. As the leader still orchestrates ordering, clients not co-located with a replica wait for 4 message delays for a response.

Detecting conflicts to boost parallelism has a long history in distributed systems [10,34,49,51]. Fast Paxos [31] was the first SMR protocol that used fast paths to minimize latency, allowing clients to contact replicas directly. Later work leveraged commutativity to increase the chances of taking the fast path [30,42,47]. As explained in §1, leader-based protocols following this approach can only resolve a conflict via a ballot change, which may require extensive state transfer between replicas and disrupts the system.

EPaxos, a leaderless SMR protocol, reduces the average latency under low conflict rates. But as we noted in §3.4 and §5.2, its tail latency is high due to a convoy effect in command execution [6, 19, 50]. It also requires clients to route requests via a data replica, increasing latency. As shown in [18], follow-ups to EPaxos, such as Caesar [3] and Atlas [17], suffer from the same problems. Tempo [18] uses a decentralized time-stamping mechanism to reduce the convoy effect in leaderless SMR, but does not fully eliminate it. Gryff [6] mixes EPaxos with ABD [4]. The protocol speeds up blind writes, but like in ABD, has expensive reads [15]. Mencius [37] distributes leader responsibilities round-robin. When conflicts are rare, this protocol is slower than EPaxos [17,38].

Several protocols rely on optimistic execution to boost performance. Speculative Paxos [44] enforces spontaneous ordering in the network to obtain identical optimistic execution at all replicas. Eve [25] executes state-machine commands optimistically in parallel, failing back to sequential execution if the results do not match. In practice, clients might not be located near a service replica [2,5]. For such clients, CURP [41] speeds up the response by executing optimistically commands at the leader. CURP does not use dependencies but instead computes a total order. In particular, a client accepts the result of optimistic execution only if it is conflict-free. SwiftPaxos is more permissive, allowing the result to be used as long as the dependency paths match.

Some protocols leverage access locality to boost SMR performance [11,12,43]. These protocols use Paxos as a black box, invoking it one or more times per command. Spanner [14] and CockroachDB [48] also use it to implement the more complex abstraction of strongly consistent transactions. All these systems may benefit from the performance improvement brought by SwiftPaxos.

Some recent works [24, 35] use dedicated network components to implement fast SMR protocols. But this approach has not yet been applied to geo-distributed systems.

7 Conclusion

Over the past decade a plethora of protocols was proposed to improve SMR in geo-distributed systems. Unfortunately, these protocols may deliver a lower performance than Paxos when contention on the replicated service increases. SwiftPaxos does not have this drawback. It executes a command in 2 message delays if there is no contention, and in 3 message delays otherwise. Our experimental evaluation demonstrates the benefits of this design. SwiftPaxos delivers 16–29% lower average latency than Paxos, and its throughput is up to 2.9x that of EPaxos.

Acknowledgements. This work was partially supported by the RACCOON project funded by the European Research Council, the PRODIGY and DECO projects funded by MCIN/AEI, and the BLOQUES project funded by the Madrid regional government. The authors thank the anonymous reviewers and their shepherd, Jay Lorch.

References

[1] SwiftPaxos codebase. `https://github.com/imdea-software/swiftpaxos`.

[2] M. S. Ardekani and D. B. Terry. A Self-Configurable Geo-Replicated Cloud Storage System. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[3] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran. Speeding up Consensus by Chasing Fast Decisions. In *Conference on Dependable Systems and Networks (DSN)*, 2017.

[4] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM*, 42(1), 1995.

[5] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Conference on Innovative Data Systems Research (CIDR)*, 2011.

[6] M. Burke, A. Cheng, and W. Lloyd. Gryff: Unifying Consensus and Shared Registers. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[7] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[8] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2), 1996.

[9] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips. Giza: Erasure Coding Objects across Global Data Centers. In *USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[10] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. *ACM Transactions on Computer Systems*, 32(4), 2015.

[11] P. R. Coelho and F. Pedone. Geographic State Machine Replication. In *Symposium on Reliable Distributed Systems (SRDS)*, 2018.

[12] P. R. Coelho and F. Pedone. GeoPaxos+: Practical Geographical State Machine Replication. In *Symposium on Reliable Distributed Systems (SRDS)*, 2021.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Symposium on Cloud Computing (SoCC)*, 2010.

[14] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-Distributed Database. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Symposium on Operating Systems Principles (SOSP)*, 2007.

[16] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2), 1988.

[17] V. Enes, C. Baquero, T. França Rezende, A. Gotsman, M. Perrin, and P. Sutra. State-Machine Replication for Planet-Scale Systems. In *European Conference on Computer Systems (EuroSys)*, 2020.

[18] V. Enes, C. Baquero, A. Gotsman, and P. Sutra. Efficient Replication via Timestamp Stability. In *European Conference on Computer Systems (EuroSys)*, 2021.

[19] T. França Rezende and P. Sutra. Leaderless State-Machine Replication: Specification, Properties, Limits. In *International Symposium on Distributed Computing (DISC)*, 2020.

[20] T. França Rezende, P. Sutra, R. Q. Saramago, and L. J. Camargos. On Making Generalized Paxos Practical. In *International Conference on Advanced Information Networking and Applications (AINA)*, 2017.

[21] M. P. Herlihy and J. M. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3), 1990.

[22] H. Howard, D. Malkhi, and A. Spiegelman. Flexible Paxos: Quorum Intersection Revisited. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2016.

[23] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference (USENIX ATC)*, 2010.

[24] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. Netchain: Scale-Free Sub-RTT Coordination. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[25] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[26] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-Data Center Consistency. In *European Conference on Computer Systems (EuroSys)*, 2013.

[27] J. Kreps, N. Narkhede, and J. Rao. Kafka: a Distributed Messaging System for Log Processing. *Workshop on Networking Meets Databases (NetDB)*, 2011.

[28] R. Ladin, B. Liskov, and L. Shrira. Lazy Replication: Exploiting the Semantics of Distributed Services. In *Symposium on Principles of Distributed Computing (PODC)*, 1990.

[29] L. Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2), 1998.

[30] L. Lamport. Generalized Consensus and Paxos. Technical report, Microsoft Research, 2005.

[31] L. Lamport. Fast Paxos. *Distributed Computing*, 19, 2006.

[32] L. Lamport, D. Malkhi, and L. Zhou. Vertical Paxos and Primary-Backup Replication. In *Symposium on Principles of Distributed Computing (PODC)*, 2009.

[33] L. Lamport and M. Massa. Cheap Paxos. In *Conference on Dependable Systems and Networks (DSN)*, 2004.

[34] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making Geo-Replicated Systems Fast as Possible, Consistent When Necessary. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[35] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[36] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *Symposium on Operating Systems Principles (SOSP)*, 1991.

[37] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machine for WANs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[38] I. Moraru, D. G. Andersen, and M. Kaminsky. There Is More Consensus in Egalitarian Parliaments. In *Symposium on Operating Systems Principles (SOSP)*, 2013.

[39] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[40] R. Pang, R. Cáceres, M. Burrows, Z. Chen, P. Dave, N. Germer, A. Golynski, K. Graney, N. Kang, L. Kissner, J. L. Korn, A. Parmar, C. D. Richards, and M. Wang. Zanzibar: Google's Consistent, Global Authorization System. In *USENIX Annual Technical Conference (USENIX ATC)*, 2019.

[41] S. J. Park and J. Ousterhout. Exploiting Commutativity For Practical Fast Replication. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[42] F. Pedone and A. Schiper. Generic Broadcast. In *International Symposium on Distributed Computing (DISC)*, 1999.

[43] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran. Making Fast Consensus Generally Faster. In *Conference on Dependable Systems and Networks (DSN)*, 2016.

[44] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Conference on Networked Systems Design and Implementation (NSDI)*, 2015.

[45] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22, 1990.

[46] A. Shraer, B. Reed, D. Malkhi, and F. P. Junqueira. Dynamic reconfiguration of primary/backup clusters. In *USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[47] P. Sutra and M. Shapiro. Fast Genuine Generalized Consensus. In *Symposium on Reliable Distributed Systems (SRDS)*, 2011.

[48] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. CockroachDB: The Resilient Geo-Distributed SQL Database. In *International Conference on Management of Data (SIGMOD)*, 2020.

[49] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Symposium on Operating Systems Principles (SOSP)*, 1995.

[50] S. Tollman, S. J. Park, and J. K. Ousterhout. EPaxos Revisited. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[51] W. E. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Transactions on Computers*, 37(12), 1988.

[52] H. Zhao, Q. Zhang, Z. Yang, M. Wu, and Y. Dai. SDPaxos: Building Efficient Semi-Decentralized Geo-replicated State Machines. In *Symposium on Cloud Computing (SoCC)*, 2018.

A Correctness

4. If at a replica $\mathsf{cmd}[id] = c \neq \bot$ then $\texttt{id}(c) = id$ and $\texttt{Propagate}(c)$ is a sent message.
5. At a replica, whenever a command c is in the START phase, it does not belong to the dependencies of any command.
6. When a replica p sends an `Ack` message, $\mathsf{cbal} = \mathsf{bal}$ at p.
7. At a ballot b
   - if $\mathsf{leader}(b)$ sends $\texttt{Ack}(b, id, D, \_)$ and $\texttt{Ack}(b, id, D', \_)$ then $D = D'$;
   - if $\mathsf{leader}(b)$ sends $\texttt{Ack}(b, id, D, \_)$ and $\texttt{Sync}(b, \_, \_, dep)$ then $dep[id] = D$;
   - if $\mathsf{leader}(b)$ sends $\texttt{Sync}(b, phase, cmd, dep)$ and $\texttt{Sync}(b, phase', cmd', dep')$ then $phase = phase'$, $cmd = cmd'$ and $dep = dep'$.
8. If a replica sends an `Ack` message at a ballot b and a $\texttt{NewLeaderAck}(b', \_, \_, \_, \_)$ message with $b' > b$ then it sends `Ack` before sending `NewLeaderAck`.
9. If the leader of a ballot b sends $\texttt{Ack}(b, \_, \_, P)$ after sending $\texttt{Ack}(b, id, D, \_)$ for $id \in Ids(P)$ then $D = Dep(id, P)$.
10. If a replica has $\mathsf{cbal} = b$, $\mathsf{dep}[id] = D$ and $id \in Accept \cup Commit$ then there is a moment when $\mathsf{leader}(b)$ has $\mathsf{cbal} = b$ and $\mathsf{status} = $ NORMAL, and after which whenever it has $\mathsf{cbal} = b$ and $\mathsf{status} = $ NORMAL it also has $\mathsf{dep}[id] = D$.
11. If $\Diamond acc(b, id, c, P)$ and $\Diamond acc(b, id, c', P')$ hold then $c = c'$ and $Dep(id, P) = Dep(id, P')$.
12. If $\Diamond acc(b, \_, \_, P)$ then for all $id \in Ids(P)$, whenever a replicas has $\mathsf{cbal} = b$ and $id \in Accept \cup Commit$, it also has $\mathsf{dep}[id] = Dep(id, P)$.
13. Whenever at a replica a command c is in the ACCEPT or the COMMIT phase, any transitive dependency of c is in the ACCEPT or the COMMIT phase as well.
14. If $\Diamond acc(b, \_, \_, P)$ holds then for all $id \in Ids(P)$ whenever at a replica we have $\mathsf{cbal} > b$, we also have $\mathsf{dep}[id] = Dep(id, P)$ and $id \in Accept \cup Commit$.

Figure 12: Additional invariants of SwiftPaxos.

In this section we prove Invariants 1–3 (defined in §3.2) together with additional low-level invariants listed in Figure 12. We then show how these invariants imply that SwiftPaxos satisfies the SMR protocol specification (§2).

Definition 1 (*acc*). We say that a command c with an identifier id is *accepted* with dependency paths P at a ballot b—and we write $acc(b, id, c, P)$—if at b there is a fast or slow quorum such that each replica of this quorum acknowledges P as the dependency paths of c with `FastAck` or `SlowAck` messages.

In the following we say that a replica sends $\texttt{Ack}(b, id, D, P)$ if it sends $\texttt{FastAck}(b, id, D, P)$ or $\texttt{SlowAck}(b, id)$ and by that time it has $\mathsf{dep}[id] = D$ and $\texttt{pset}(id) = P$.

We write $\Diamond acc(b, id, c, P)$ to denote that at some point of protocol execution $acc(b, id, c, P)$ holds.

Definition 2 (*Ids*). We say that *id appears* in a set of dependency paths P (or simply *id in* P) if there exists at least one path in P that contains id. We define $Ids(P)$ as the set of all id in P.

Definition 3 (*Dep*). For a set of dependency paths P and $id \in Ids(P)$, we define $Dep(id, P)$ as the dependency set of id inferred from P.

Note that by definitions of acc and dependency paths, if $\Diamond acc(b, id_0, c, P)$ holds for some quorum Q then at the moment when a replica $q \in Q$ sends $\texttt{Ack}(b, id_0, \_, P)$ message it has $\mathsf{dep}[id] = Dep(id, P)$ for all $id \in Ids(P)$.

Invariants 4–10 easily follow from the structure of the protocol. We now prove the rest of the invariants.

Proof of Invariant 11. Each time a command c is accepted at a ballot the leader of this ballot must acknowledge c's dependencies either with an `Ack` or a `Sync` message. By Invariant 7 the leader cannot acknowledge two distinct dependency sets for the same command at a single ballot. □

Proof of Invariant 12. Assume $\Diamond acc(b, id_0, \_, P)$ and let id be any identifier in $Ids(P)$. Take a replica p with $\mathsf{cbal} = b$ and $id \in Accept \cup Commit$. Two cases are possible: either p receives $\texttt{Ack}(b, id, D, \_)$ from $\mathsf{leader}(b)$, or it receives `Sync` carrying id in the ACCEPT or COMMIT phase.

First assume that p receives $\texttt{Ack}(b, id, D, \_)$ from the leader of b. Since $\Diamond acc(b, id_0, \_, P)$ holds, the leader also sends $\texttt{Ack}(b, id_0, \_, P)$. Then by Invariant 9, we have $D = Dep(id, P)$. Hence, when p receives $\texttt{Ack}(b, id, D, \_)$ from $\mathsf{leader}(b)$ it sets $\mathsf{dep}[id]$ to $D = Dep(id, P)$ and after that—according to Invariant 7—$\mathsf{dep}[id]$ remains unchanged at b, as required.

Suppose now, that p receives `Sync` carrying id in the ACCEPT or COMMIT phase with the dependency set D. Then any replica with $\mathsf{cbal} = b$ receives that message, sets $\mathsf{dep}[id]$ to D and by Invariant 7 never changes it at b. When $\mathsf{leader}(b)$ sends $\texttt{Ack}(b, id_0, \_, P)$ message it has $\mathsf{dep}[id] = Dep(id, P) = D$, as required. □

Proof of Invariant 13. We prove the invariant by induction on the length of the protocol execution. Initially at any replica each command is in the START phase and hence, the invariant is satisfied. We now consider transitions at lines 22, 30, and 48, as they are the only transitions affecting the validity of the invariant in a nontrivial way.

- Transition at line 22. According to line 23 after the execution of the handler at line 22 for a command c, any dependency of c is in the ACCEPT or COMMIT phase and hence, by induction hypothesis any transitive dependency of c satisfies the invariant.
- Transition at line 30. The handler simply moves an ACCEPTed command to the COMMIT phase, which allows us to conclude with the induction hypothesis.
- Transition at line 48. By the structure of the protocol after a replica p executes this handler all the commands are in the START, ACCEPT or COMMIT phase at p. From Invariant 5 the commands that are in the START phase do not belong to the dependency set of any command, hence our property.

□

Proof of Invariant 3. By the induction on the length of the protocol execution: at the beginning of the execution dep is empty at each replica, which makes the dependency graphs acyclic; for the induction step we consider transitions at lines 6, 22, and 48—the only transitions affecting the validity of the invariant in a nontrivial way.

- Transition at line 6. In order to introduce a cycle upon receipt of a $\texttt{Propagate}(c)$ message at a replica p, $\texttt{id}(c)$ must belong to the dependencies of some other command before the transition is triggered. However, according to the precondition at line 7 at this moment c is in the START phase and, thus, from Invariant 5 no command depends on it, as required.
- Transition at line 22. We prove this case by contradiction. Assume that right after a replica p executed the handler at line 22 for a command c, some command transitively depends on itself at p. According to Invariants 10 and 13, and the precondition at line 23, all commands of this cycle are in sync with the leader at the time when the leader sends `Ack` for c. However, by induction hypothesis, by that time the dependency graph at the leader is acyclic, which yields a contradiction.
- Transition at line 48. According to line 64, after a replica executes the handler at line 48 its dependency graph is acyclic, as required.

□

The proof of Invariant 2 relies on the following proposition:

Proposition 1. Whenever at a replica p we have $id \notin p.\mathsf{dep}[id']$ and $id' \notin p.\mathsf{dep}[id]$ for two identifiers id and id' of two conflicting commands:

1. p is not the leader of its ballot (i.e., $p \neq \mathsf{leader}(p.\mathsf{cbal})$);
2. at p one of two identifiers is in the PREACCEPT phase while the other is in the ACCEPT or the COMMIT phase.

Proof. We prove the proposition by induction on the length of the protocol execution. The property holds at the beginning of the execution because no commands are recorded at any of replicas. For the induction step we consider transitions at lines 6, 22, and 48, as they are the only transitions affecting the validity of the proposition in a nontrivial way.

- Transition at line 6. Consider a moment when a replica p receives a $\texttt{Propagate}(c')$ message and take any command c that conflicts with c' and which is already recorded in $p.\mathsf{cmd}$. According to line 11, after the execution of handler at line 6 we have $\texttt{id}(c) \in p.\mathsf{dep}[\texttt{id}(c')]$.

- Transition at line 22. Assume that p receives a `FastAck`$(b, id, D, \_)$ message for a command c from the leader. Take any command conflicting with c with an identifier id' that is already recorded at p at the moment of the execution of transition. Let D' be $p.\mathsf{dep}[id']$ at this moment.

  If p is the leader of b then it receives its own proposal, and thus the handler is not affecting dep at p. By induction hypothesis, before executing this transition process p has $id \in p.\mathsf{dep}[id']$ or $id' \in p.\mathsf{dep}[id]$, as required.

  Suppose that $p \neq \mathsf{leader}(b)$. If by the moment p executes line 22 it has $\mathsf{phase}[id'] = \text{PREACCEPT}$ then the transition does not affect the validity of the proposition because in this case p moves id to the ACCEPT phase without changing $\mathsf{phase}[id']$. Thus, p has $\mathsf{phase}[id] = \text{ACCEPT}$ and $\mathsf{phase}[id'] = \text{PREACCEPT}$ after executing the transition, as required. If $\mathsf{phase}[id'] \in \{\text{ACCEPT}, \text{COMMIT}\}$ then after the execution of the transition both identifiers are in the ACCEPT or the COMMIT phase. Therefore, from Invariant 10 there is a moment when $\mathsf{leader}(b)$ has $\mathsf{dep}[id] = D$ and $\mathsf{dep}[id'] = D'$. By induction hypothesis at this moment $\mathsf{leader}(b)$ has $id \in \mathsf{dep}[id']$ or $id' \in \mathsf{dep}[id]$, as required.
- Transition at line 48. Let id and id' be two identifiers of conflicting commands such that process p has $\mathsf{dep}[id] = D \neq \bot$ and $\mathsf{dep}[id'] = D' \neq \bot$ after executing transition at line 48. By the structure of the algorithm $id \in \mathit{Accept} \cup \mathit{Commit}$ and $id' \in \mathit{Accept} \cup \mathit{Commit}$ (lines 54 and 59). We prove that either $id \in \mathsf{dep}[id']$ or $id' \in \mathsf{dep}[id]$. We know that both identifiers satisfy condition at line 53 or condition at line 58. If both identifiers satisfy condition at line 53 then at some point $\mathsf{leader}(b_{\max})$ has $\mathsf{dep}[id] = D$ and $\mathsf{dep}[id'] = D'$ (Invariant 10). By induction hypothesis by this moment either $id \in D'$ or $id' \in D$, as required. Assume now that at least one command satisfies condition at line 58. According to line 61 either $id \in \mathsf{dep}[id']$ or $id' \in \mathsf{dep}[id]$.

□

Proof of Invariant 2. Whenever two conflicting commands are committed at a replica p both of them are in the COMMIT phase, which by Proposition 1 means that one of the two is a dependency of the other. □

Proposition 2. If $\Diamond acc(b, \_, \_, P)$ then whenever there is a quorum Q such that each replica $q \in Q$ sends the `NewLeaderAck`$(b', cbal_q, phase_q, \_, dep_q)$ message with $b' > b$ and $cbal_q \leq b$, the following property holds for any $id \in Ids(P)$:

> There is a quorum $Q' \in \mathcal{FQ}(b) \cup \mathcal{SQ}(b)$ such that for any replica $q \in Q \cap Q'$ we have $dep_q[id] = Dep(id, P)$, $cbal_q = b$ and if $Q' \in \mathcal{SQ}(b)$ then $\forall q \in Q \cap Q'. phase_q[id] \in \{\text{ACCEPT}, \text{COMMIT}\}$.

Proof. Assume $\Diamond acc(b, id_0, \_, P)$ and let Q be a quorum such that each replica $q \in Q$ sends a `NewLeaderAck`$(b', cbal_q, phase_q, \_, dep_q)$ message with $b' > b$ and $cbal_q \leq b$. Let $id \in Ids(P)$. Since $\Diamond acc(b, id_0, \_, P)$, there exists a quorum $Q' \in \mathcal{FQ}(b) \cup \mathcal{SQ}(b)$ such that each replica in Q' sends `Ack`$(b, id_0, \_, P)$ message, and by that time it has $\mathsf{dep}[id] = Dep(id, P)$. After sending `Ack`s, replicas in Q' can change $\mathsf{dep}[id]$ at b only if they receive an `Ack` message from $\mathsf{leader}(b)$ with a proposal different from D. However, according to Invariant 7, once $\mathsf{leader}(b)$ has voted for D it never sends a new proposal at b. Thus, after sending `Ack`, replicas in Q' never change $\mathsf{dep}[id]$ at b. Moreover, from Invariant 8 we know that at the moment a replica in Q' sends a `NewLeaderAck` message it has $\mathsf{cbal} \geq b$ and since we assume that for all $q \in Q$ we have $cbal_q \leq b$ we also have $\forall q \in Q \cap Q'$, $cbal_q = b$. Hence, when a replica $q \in Q \cap Q'$ sends the `NewLeaderAck` message, it has $dep_q[id] = Dep(id, P)$, as required. If now we assume that $Q' \in \mathcal{SQ}(b)$ then all replicas in Q' send `SlowAck`(b, id) messages, implying that for all $q \in Q \cap Q'$, $phase_q[id] \in \{\text{ACCEPT}, \text{COMMIT}\}$ (lines 24 and 75). □

Proof Invariant 14. We prove the invariant by induction on the value of cbal. Assume $\Diamond acc(b, id_0, \_, P)$ and consider a ballot $b' > b$ such that

> For all $id \in Ids(P)$, whenever a replica has $b < \mathsf{cbal} < b'$, it also has $\mathsf{dep}[id] = Dep(id, P)$ and $id \in \mathit{Accept} \cup \mathit{Commit}$. (H)

We prove by induction on the length of the protocol execution that

> For all $id \in Ids(P)$, whenever a replica has $\mathsf{cbal} = b'$, it also has $\mathsf{dep}[id] = Dep(id, P)$ and $id \in \mathit{Accept} \cup \mathit{Commit}$. (1)

At the beginning of the execution at each replica we have $\mathsf{cbal} = 0 \leq b < b'$, hence (1) trivially holds. For the induction step, only handler at line 48 affects validity of the invariant in a nontrivial way. Let p be a replica that receives a `NewLeaderAck`$(b', cbal_q, phase_q, \_, dep_q)$ message from each $q \in Q$. Let $b_{\max} = \max\{cbal_q \mid q \in Q\}$ and $U = \{q \in Q \mid cbal_q = b_{\max}\}$. We prove that after executing transition at line 48, p has $\mathsf{dep}[id] = Dep(id, P)$ and $id \in \mathit{Accept} \cup \mathit{Commit}$ for all $id \in Ids(P)$.

We first note that $b_{\max} \geq b$. Indeed, from Invariant 8 replicas in Q that have voted for $Dep(id, P)$ at b send an `Ack` message before `NewLeaderAck`, hence $b_{\max} \geq b$.

As the first step, we prove that after executing transition at line 48 up to line 61, p has $\mathsf{dep}[id] = Dep(id,P)$ and $id \in Accept \cup Commit$ for any $id \in Ids(P)$. We then prove that after executing lines 62–64 the dependency sets of all identifiers in $Ids(P)$ remain unchanged.

Step 1. Take any $id \in Ids(P)$ and c such that $\mathtt{id}(c) = id$.

- Suppose that $b_{\max} > b$. By (H), id satisfies condition at line 53. Process p sets $\mathsf{dep}[id]$ to $Dep(id,P)$ and $\mathsf{phase}[id]$ to ACCEPT or COMMIT, as required.
- Suppose now that $b_{\max} = b$ (i.e., $\forall q \in Q.\, cbal_q \leq b$). From Proposition 2, either the condition at line 53 or the one at line 58 is satisfied for id. Assume that the condition at line 53 holds: there exists a replica $q \in Q$ such that $cbal_q = b$ and $phase_q[id] \in \{\text{ACCEPT}, \text{COMMIT}\}$. Then at line 56 process p sets $\mathsf{dep}[id]$ to $dep_q[id] = Dep(id,P)$ (Invariant 12) and $\mathsf{phase}[id]$ to $phase_q[id]$, as required. Assume now, that the condition at line 58 holds. Then for all $q \in U$ we have $phase_q[id] \notin \{\text{ACCEPT}, \text{COMMIT}\}$ and there exists a quorum $Q' \in \mathcal{FQ}(b_{\max})$ such that $Q \cap Q' \subseteq U$ and all replicas in $Q \cap Q'$ report the same dependency set D for id. From Proposition 2, there exists a quorum Q_0 such that $Q \cap Q_0 \subseteq U$ and that for any replica in $q \in Q \cap Q_0$, $dep_q[id] = Dep(id,P)$. Since $Q \cap Q' \cap Q_0 \neq \emptyset$, we have $D = Dep(id,P)$. We now prove by contradiction that $\mathsf{dep}[id]$ computed at line 61 equals D. Assume the converse. Then there exists a command conflicting with c with an identifier id' such that $id' \notin D$ and by the time p executes line 61 for id it has $id \notin \mathsf{dep}[id']$ and $id' \notin Start$. Let D' be $\mathsf{dep}[id']$ at that moment. Since $id' \notin Start$ either condition at line 53 or the one at line 58 is satisfied for id' with D'.
  (a) Assume that condition at line 53 is satisfied. There exists a replica $q \in U$ such that $phase_q[id'] \in \{\text{ACCEPT}, \text{COMMIT}\}$ and $dep_q[id'] = D'$, with $id \notin D'$. From Invariant 10 there is a moment t after which whenever $\mathsf{leader}(b)$ has $\mathsf{cbal} = b$ it also has $\mathsf{dep}[id'] = D'$. Moreover, there is also a moment t' when $\mathsf{leader}(b)$ sends the $\mathtt{Ack}(b, id_0, \_, P)$ message, and by this moment it has $\mathsf{dep}[id] = Dep(id,P) = D$. Therefore, at $\max(t,t')$, the leader of b has $\mathsf{dep}[id] = D$ and $\mathsf{dep}[id'] = D'$, with $id \notin D'$ and $id' \notin D$, contradicting Proposition 1.
  (b) Suppose that the condition at line 58 is satisfied for id': there exists a fast quorum Q'' such that $Q \cap Q'' \subseteq U$ and all replicas in $Q \cap Q''$ report D' as the dependency set of id'. Since $Q \cap Q'' \cap Q_0 \neq \emptyset$, there is a replica in $q \in Q \cap Q''$ with $dep_q[id] = Dep(id,P)$ and $dep_q[id'] = D'$. Recall that we are under the assumption that both identifiers, id and id', satisfy condition at line 58. Therefore, they also satisfy loop condition at line 57. As a result we have $\forall q' \in U.\, phase_{q'}[id] \notin \{\text{ACCEPT}, \text{COMMIT}\} \wedge phase_{q'}[id'] \notin \{\text{ACCEPT}, \text{COMMIT}\}$. Thus, $phase_q[id] = \text{PREACCEPT}$ and $phase_q[id'] = \text{PREACCEPT}$, contradicting Proposition 1.

  We thus have proved that $\mathsf{dep}[id]$ computed at line 61 equals $D = Dep(id,P)$, as required.

Step 2. Now that we have proved that after executing transition at line 48 up to line 61, p has $\mathsf{dep}[id] = Dep(id,P)$ for all $id \in Ids(P)$, we prove that the loop at line 62 does not affect these dependencies. Let $id \in Ids(P)$ and let id' be any identifier in $\mathsf{dep}[id]$ at p at the moment when p executes lines 62–63. Since at this moment $\mathsf{dep}[id] = Dep(id,P)$, we have $id' \in Ids(P)$ (by definition of Ids). As we saw earlier, either $b_{\max} = b$ or $b_{\max} > b$ and according to Proposition 2 and (H), id' either satisfies condition at line 53 or condition at line 58. Thus, at the moment p executes lines 62–63 it has $id' \notin Start$ and hence condition at line 62 does not hold for (id, id'), as required.

Step 3. Finally we prove by contradiction that after executing line 64, p has $\mathsf{dep}[id] = Dep(id,P)$ for all $id \in Ids(P)$. Assume that at p, just before the execution of line 64 there is a cycle $id_1 \to \cdots \to id_n \to id_1$ in the dependency graph. We prove by contradiction that none of the identifiers of this cycle belongs to $Ids(P)$. Assume the converse, i.e., there exists $m \in [1;n]$ such that $id_m \in Ids(P)$. We have already proved that prior to executing line 64 process p has $\mathsf{dep}[id] = Dep(id,P)$ for all $id \in Ids(P)$. Thus $id_{m-1} \in Dep(id_m, P)$ also belongs to $Ids(P)$. Moreover, all identifiers of the cycle are in $Ids(P)$ because each of them is a transitive dependency of each other. Hence, when $\mathsf{leader}(b)$ sends an $\mathtt{Ack}(b, id_0, \_, P)$ message it has this cycle in its dependency graph, contradicting Invariant 3.

□

Proof of Invariant 1. Take two replicas that commit a command c with an identifier id. We have $\Diamond acc(b,id,c,P)$ and $\Diamond acc(b',id,c,P')$ for some b and b'. Let D and D' be $Dep(id,P)$ and $Dep(id,P')$, respectively. We prove that $D = D'$. Assuming that b equals b' we conclude with Invariant 11. Suppose now that $b \neq b'$ and (without loss of generality) that $b < b'$. Consider the moment when a replica p sends $\mathtt{Ack}(b', id, D', P')$. From Invariant 6 at this moment $\mathsf{cbal} = b' > b$ at p. Hence, from Invariant 14 we have $D = D'$, as required. □

A.1 Validity

Proof of Validity. Consider a replica that executes a command c. By Invariant 4, $\mathtt{Propagate}(c)$ is a sent message, hence, some client submitted c. □

A.2 Integrity

Proof of Integrity. At a replica, each executed command is tracked with the set Exec: a command can be executed only if its identifier is not included in the set, and once it is executed, the set is updated to include the identifier. Furthermore, during the protocol execution no elements are removed from Exec and only one identifier can be associated with a single command. Hence, at each replica line 34 is executed at most once per command. ☐

A.3 Ordering

To prove Ordering we first introduce the following notations: for two commands c and d we write (i) $c \lhd_p d$ if $c \bowtie d$ and some replica p executes c before d, and (ii) $c \blacktriangleleft d$ if c was executed by some replica before any process submitted d. Using these notations the relation $\prec$ from §2 can be expressed as $\prec = (\bigcup_p \lhd_p) \cup \blacktriangleleft$.

Proposition 3. If for two conflicting commands c and d we have $c \lhd_p d$ for some replica p then at the moment when p executes c, we have $\texttt{id}(d) \notin \texttt{dep}[\texttt{id}(c)]$.

Proof. Consider the moment when p executes c. The Integrity property and $c \lhd_p d$ imply that p has not executed d at this moment, and thus $\texttt{id}(d) \notin$ Exec at p. From the precondition at line 33, $\texttt{dep}[\texttt{id}(c)] \subseteq$ Exec, so that $\texttt{id}(d) \notin \texttt{dep}[\texttt{id}(c)]$. ☐

Proposition 4. The relation $\bigcup_p \lhd_p$ is asymmetric.

Proof. Assume the contrary: $c \lhd_p d$ and $d \lhd_q c$ for two conflicting commands c and d. From Proposition 3, when p executes c it has $\texttt{id}(d) \notin \texttt{dep}[\texttt{id}(c)]$, and when q executes d it has $\texttt{id}(c) \notin \texttt{dep}[\texttt{id}(d)]$. But this contradicts Invariant 2. ☐

Proposition 5. Assume $c_1 \prec c_2$. Whenever a replica p executes c_2, some replica has already executed c_1.

Proof. Assume $c_1 \prec c_2$ and that replica p executes c_2. Then we have either $c_1 \blacktriangleleft c_2$ or $c_1 \lhd_q c_2$ for some replica q. Consider first the case when $c_1 \blacktriangleleft c_2$. Then c_1 is executed at some replica before c_2 is submitted and, hence, before c_2 is executed at p, as required. Consider now the case when $c_1 \lhd_q c_2$ for some replica q. We must have either $c_1 \lhd_p c_2$ or $c_2 \lhd_p c_1$. The latter case would contradict Proposition 4, so that $c_1 \lhd_p c_2$, as required. ☐

Proposition 6. Assume $c_1 \prec \cdots \prec c_n$ for $n \geq 2$. Whenever a replica p executes c_n, some replica has already executed c_1.

Proof. Follows from Proposition 5 by induction on n. ☐

Proof of Ordering. By contradiction, assume that $c_1 \prec \cdots \prec c_n = c_1$ for $n \geq 2$. Consider the first time when some replica executes c_1. By Proposition 6, at this time c_1 has already executed at some replica, which yields a contradiction. ☐

A.4 Liveness

For simplicity, in this section we assume that the set of slow quorums consists of all majorities, i.e., $\forall b.\, \mathcal{SQ}(b) = \{Q \in \mathcal{P}(\mathcal{R}) \mid |Q| \geq \lceil N/2 \rceil\}$. Since we assume that the system is eventually synchronous (§2), it is possible to implement an *eventually perfect failure detector*, denoted $\diamond P$ [8]. At each replica p, the failure detector $\diamond P$ outputs the set of replicas that p believes are correct. $\diamond P$ guarantees that eventually all crashed replicas are detected by p and no correct replica remains indefinitely suspected.

```
77 Alive ← $\mathcal{R} \subseteq \mathcal{R}$
78 trusted ← leader(0) $\in \mathcal{R}$
79 when $\diamond P \neq$ Alive
80     Alive ← $\diamond P$
81     trusted ← select(Alive)
82 when leader(bal) $\neq$ trusted
83     if $p$ = trusted then recover()
84     else send Recover(bal) to trusted
85 when received Recover($b$)
86     if $p$ = trusted then recover()
```

Figure 13: Recovery policy at a replica p.

In Figure 13 we present an algorithm according to which a replica p invokes the recovery mechanism. $\diamond P$ constantly outputs a set of correct replicas into a variable Alive (line 80), which is then used to select the trusted replica (line 81), i.e., the replica that (according to p) should be the leader. To this end, p passes Alive to a function `select` that deterministically picks one replica among the set of replicas. A possible implementation of `select` returns the replica with the lowest identifier. A replica constantly checks if its leader is the trusted one (line 82). If this is not the case p checks whether it trusts itself—in which case p initiates recovery (line 83)—or some other replica—which causes p to send a `Recover` message to trusted (line 84). With this message p asks trusted to impose its leadership: upon the receipt of $\texttt{Recover}(b)$, a replica verifies whether it should be a leader and then starts the recovery (line 86). The `Recover` message carries the value of bal, which helps trusted to choose at line 41 a ballot high enough to convince p to follow it: when p receives $\texttt{Recover}(b)$ it selects a ballot b' owned by p such that $b' > b$. Additionally, if possible, trusted tries to choose b' such that all replicas of at least one fast quorum associated with b' belong to Alive, i.e., $\exists Q \in \mathcal{FQ}(b').\, Q \subseteq$ Alive.

We now describe the measures to be taken on the client-side of the protocol. First of all, a submitted command may be lost during the recovery if it has not been accepted before. For example, suppose that at some ballot b a replica p receives a command c submitted by a correct client. Then p starts a new ballot b' and becomes its leader. If c is not accepted at b before the recovery is started, p might not include c in the new state (conditions at lines 53 and 58 do not hold for c). Since p already received c, it will never make a new proposal for c at b' or any subsequent ballots. To avoid such scenarios, we allow clients to resubmit their requests. Note that this does not affect the proof of Integrity. In the example above, the leader receives c at b' or any higher ballot and eventually makes its proposal.

Finally, as discussed in §4.3 an additional mechanism is needed to deal with client failures. A client may crash at any time and thus may fail to send a command to a subset of the correct replicas. If for each fast and slow quorum only some replicas receive request c then further progress is impossible not only for c but also for any command c' that depends on c at the leader. To circumvent such situations, a replica tracks the command identifiers that belong to the dependency sets received in leader's `FastAck` messages. If the replica detects that the payload of one of such commands is missing, it asks its leader to retransmit it. It is enough to contact the leader because no command is committed if it is not received by the leader. It is also possible that some followers receive a command c while the leader does not. These followers will take a dependency on c for any conflicting command, disabling a fast path for each of them. To deal with this, followers re-propose c if it is not committed after some timeout.

To prove that the above mechanisms ensure Liveness we rely on the following propositions.

Proposition 7. Either no correct replica ever receives a `NewLeader` message, or there exist a correct replica p and a ballot b owned by p such that

1. each correct replica eventually trusts only p;
2. b is the maximal ballot received by any correct replica in `NewLeader` messages.

Proof. If no correct replica ever receives a `NewLeader` message then the proposition trivially holds. Assume that at least one correct replica receives a `NewLeader` message. From the properties of $\Diamond P$, there exist a replica p and a time t_0 such that after this time failures stop occurring and at each correct replica $\mathsf{trusted} = p$ and Alive is the set of all correct replicas.

We first prove by contradiction that p sends a finite number of `NewLeader` messages. Assume the converse. After t_0, p is the only correct replica that can start the recovery. Therefore, there is only a finite number of the `NewLeader` messages sent by the correct replicas other than p, because all such messages are sent before t_0. Moreover, the number of messages sent by a faulty replica is finite by definition. Thus, each correct replica joins only a finite number of ballots that are not owned by p. Since p increases its ballot with each issued `NewLeader` message and because the number of such messages is infinite, there is a time $t_1 \geq t_0$ after which each correct replica always follows p (i.e., $\mathsf{leader}(p.\mathsf{bal}) = p$). Therefore, at t_1, there are no replica at which the condition at line 82 is satisfied. Hence, no replica sends a `Recover` message after t_1 and thus the total number of such messages is finite. Let $t_2 \geq t_1$ be the time after which p no longer receives `Recover` messages. After t_2, p never executes handler at line 85. Therefore, after t_2, p never executes lines 83 or 86, and hence, it stops sending the `NewLeader` messages, which contradicts our assumption.

Now, that we have proved that there is only a finite number of `NewLeader` messages sent by p, let b be the maximal ballot for which p sends a `NewLeader` message. We prove by contradiction that for each `NewLeader`(b') message received by a correct replica, $b' \leq b$. Assume the contrary. Let q be a correct replica receiving at time t' the `NewLeader`(b') message from a replica $p' \neq p$ with $b' > b$. After t', q has $\mathsf{bal} \geq b'$. Since p never chooses a ballot greater than $b < b'$, after t' we also have $\mathsf{leader}(q.\mathsf{bal}) \neq p$. Eventually q trusts p and thus at some point it forces p to recover (line 82) with the ballot at least as high as b' (lines 83 and 84). As a result, p sends the `NewLeader`(b'') message with $b'' > b' > b$, contradicting the fact that b is the maximal ballot chosen by p during the whole execution. □

Proposition 8. There exist a ballot b and a point in time after which every correct replica has $\mathsf{status} = \mathsf{NORMAL}$ and $\mathsf{bal} = \mathsf{cbal} = b$.

Proof. If during the whole execution no correct replica ever receives a `NewLeader` message then all correct replicas always follow the same correct replica at the same ballot, as required. If at least one correct replica receives a `NewLeader` message then according to Proposition 7, there exists a correct replica p eventually trusted by all correct replicas and a ballot b such that b is the maximal ballot received by a correct replica in `NewLeader` messages. Eventually a correct replica q receives the `NewLeader`(b) message and joins b. After that, q never changes its ballot, as it ignores any other `NewLeader` messages (line 44). All correct replicas eventually answer to p with a `NewLeaderAck` message. Thus, eventually p sends `Sync` to all correct replicas, as required. □

Proposition 9. If a replica p commits c with a dependency set D then any correct replica eventually commits c with D.

Proof. Suppose that a replica p commits c with D at a ballot b. From Proposition 8 we know that there exists a ballot b' after which the system stabilizes with the one correct leader.

We first prove by contradiction that $b \leq b'$. Suppose that $b > b'$. Since c is committed at b, there is at least one correct replica q that sends an `Ack` message for c at b. To this end, q has to join b (i.e., at some point we have $q.\mathsf{cbal} = b$), however, $b' < b$ is the maximal ballot joined by q (Proposition 8), hence the contradiction.

We thus have $b \leq b'$. Suppose that $b = b'$. This means that there is a quorum of correct replicas sending `Ack` messages that lead to acceptance of c. Eventually each correct replica receives all these messages and commits c with D.

Suppose now that $b < b'$. Eventually each correct replica joins b' (i.e., $\mathsf{cbal} = b'$). In order to join b', each correct replica receives the $\mathtt{Sync}(b',\_,\_,\_)$ message, sent by $\mathsf{leader}(b')$. Moreover, according to Invariant 14 this message carries a state that contains c with the dependencies D. □

Proof of Liveness. We first prove that if a command c is executed by a replica p then c is eventually executed by all correct replicas. Before executing c, p commits c together with c's transitive dependencies D. According to Proposition 9, the correct replicas eventually commit c as well as any command in D, which enables the condition at line 33 for c.

We now prove that if a command c is submitted by a non-faulty client then c is eventually executed by all correct replicas. Let b be the last stable ballot (Proposition 8). Note that the client submitting c is correct and that it resubmits c if it does not receive a response after some time. Hence, eventually c is accepted at some ballot. Let D be the set of all transitive dependencies of c. Similarly to c, those commands in D that are submitted by non-faulty clients are eventually accepted. Moreover, from Invariant 13 and Invariant 14, when c is accepted and the leader of b has $\mathsf{cbal} = b$, it also has payload of any command in D. Therefore, by contacting the leader and thus obtaining the missing payloads according to the mechanics described at the beginning of the section, the replicas eventually accept the commands in D submitted by the faulty clients. Regardless of the ballots at which the different commands in D are accepted, from Invariant 14, each correct replica eventually receives and commits c together with any command in D. We conclude with Invariant 3 according to which, once committed, c is never blocked in the execution. □

A.5 Nontriviality

The proof of Nontriviality relies on the following proposition:

Proposition 10. If $\Diamond acc(b,\_,\_,P)$ then for all $id \in Ids(P)$ all correct replicas eventually commit id with $Dep(id,P)$.

Proof. From Proposition 8 we know that there exist a ballot $b' \geq b$ and a time t after which recovery stops occurring and at least a majority of replicas are correct. If $b' = b$ then by the structure of the protocol all correct replicas eventually commit $id \in Ids(P)$ with $Dep(id,P)$—this is because communication between replicas cannot be disturbed by recovery or failures and because replicas communicate through reliable channels.

Assume that $b' > b$. Eventually each correct replica receives a $\mathtt{Sync}(b',\_,\_,\_)$ message carrying id with $Dep(id,P)$ and $\mathsf{phase}[id] = \text{COMMIT}$ or $\mathsf{phase}[id] = \text{ACCEPT}$ (Invariant 14). If $\mathsf{phase}[id] = \text{COMMIT}$, a correct replica commits id immediately after receiving `Sync`, as required. If $\mathsf{phase}[id] = \text{ACCEPT}$, correct replicas of some slow quorum eventually broadcast a $\mathtt{SlowAck}(b',id)$ message (line 76). Eventually replicas at b' receive `SlowAcks` from each replica of this quorum and commit id with $Dep(id,P)$, as required. □

Proof of Nontriviality. A client accepts the result of execution of a command c only if there is a ballot b at which $\mathsf{leader}(b)$ optimistically executes c and replicas of some quorum of b agree on c's dependency paths P. Thus, if the client gets the result of c then we have $\Diamond acc(b,\_,c,P)$. Note that $\mathsf{leader}(b)$ executes client requests in the order inferred form P, i.e., for each $id \in Ids(P)$, $\mathsf{cmd}[id]$ is executed with respect to dependency set $Dep(id,P)$. Moreover, according to Proposition 10, each correct replica eventually commits $id \in Ids(P)$ with $Dep(id,P)$, and hence, it eventually executes c following the same order. Thus, the result of execution of c at any correct replica equals to the result of optimistic execution at $\mathsf{leader}(b)$, as required. □

A.6 Latency

Asynchrony and failures may arbitrarily delay the execution of a command. For that reason, we measure latency only after the system stabilizes. We denote by δ the upper bound on the message delay after stabilization (see §2). As usual, we ignore the cost of local computation (in our setting, it is orders of magnitude lower than latency). The latency of a protocol is the maximum amount of time a client must wait before delivering a response once the system is stable.

Table 1 summarizes the latency of various state-machine replication protocols. We consider the following classes of runs commonly found in practice: *(sequential)* commands are never concurrent, i.e., upon the submission of a command, prior submitted commands are already committed everywhere; *(conflict-free)* concurrent commands are all commuting; and *(contention-free)* all

	sequential	*conflict-free*	*contention-free*	*general*
Paxos [29]	4δ			
N^2Paxos [29]	3δ+1			
Mencius [37]	2δ+1	4δ+1		
FastPaxos+ [31]	2δ+1	3δ+1		
Generalized Paxos [30]	2δ+1			6δ+1
Egalitarian Paxos [38]	2δ+1			$O(n\delta)$
CURP+N^2Paxos [41]	2δ		3δ+1	
SwiftPaxos	2δ			3δ

Table 1: Latency of state-machine replication protocols. We denote by δ the upper bound on message delay when the system is stable. For non-colocated clients +1 denotes the additional message delay.

concurrent conflicting commands are received in the same order at the replicas. The last class *(general)* is the worst-case latency when the system is stable. For protocols that have a fast path this corresponds to the slow path. Worst-case latency matters for hot items and when the fast path is no longer available (e.g., if a machine gets disconnected). Notice that these classes are ordered as follows: sequential $\subsetneq$ conflict-free $\subsetneq$ contention-free $\subsetneq$ general. In Table 1, the notation +1 refers to the additional message delay to reach non-colocated clients.

Below, we prove that the latency of SwiftPaxos matches the results in Table 1. Namely, we establish that SwiftPaxos executes state-machine commands in two message delays in the absence of contention, and three otherwise.

Lemma 1. Assume a contention-free run ρ of SwiftPaxos. At any time t in ρ, if p and q in some $\mathcal{FQ}(b)$ have both received c at t, then *(i)* $\mathsf{dep}[c]$ is the same at the two processes, and *(ii)* the evaluation of $\mathtt{pset}(id(c))$ on the two processes is identical.

Proof. We prove this result inductively. The property holds obviously at $t = 0$. Assume that it is true at $t-1$. At time t, if no command is newly received by either p or q, we are done. Otherwise, let c be such a command and wlog. consider that q receives c at time t. Process q executes line 11 for command c at that time. Consider a command d in $\mathsf{dep}_p[c]$ at t. As the run is contention-free, d is received before time t at q. Thus, it is added to $\mathsf{dep}_q[c]$ at t. This shows that $\mathsf{dep}_p[c] \subseteq \mathsf{dep}_q[c]$ at t. Using a symmetrical argument, we have $\mathsf{dep}_q[c] \subseteq \mathsf{dep}_p[c]$, as required. Now consider that q evaluates $\mathtt{pset}(id(c))$ to the set of paths P at t. By a short induction, because for any received command d, $\mathsf{dep}_p[d] = \mathsf{dep}_q[c]$ at t, the evaluation on p must also return P. □

Proposition 11. The latency of SwiftPaxos is 2δ when the run is contention-free, and 3δ otherwise.

Proof. Consider a run ρ of SwiftPaxos. Let t_0 be the point in time after which the system is stable in ρ. By Proposition 8, for some ballot b_0 at every correct replica, $\mathsf{bal} = \mathsf{cbal} = b_0$ and $\mathsf{status} = \text{NORMAL}$ hold forever after time t_0. Let p be the leader of ballot b_0. Pick a command c submitted after time t_0. According to SwiftPaxos (line 2), $\mathtt{client}(c)$ broadcasts a message $\mathtt{Propagate}(c)$ to all replicas. Any replica receives such a message after δ units When receiving it, p sends a message $\mathtt{Reply}(\mathsf{bal}, \mathtt{id}(c), P, r)$ to $\mathtt{client}(c)$, with r the result of the optimistic execution of c (line 17). There are two cases to consider:

- *(ρ is contention-free)* Assume a command d non-commuting with c was received at replica p before c. Because c is uncontended, it must be the case that d was also received before c at any other replica $q \in \mathcal{FQ}(b_0)$. By Lemma 1, p and q execute the exact same computation at lines 11 and 12. From which, we deduce that all the fast quorum replicas broadcast the same message $\mathtt{FastAck}(b_0, \mathtt{id}(c), D, P)$ to $\mathtt{client}(c)$ (lines 19 and 21). The precondition at line 3 on $\mathtt{client}(c)$ triggers after (at most) δ units of time, delivering the response of c. It follows that the message delay of c is 2δ in this case.
- *(Otherwise)* Leader p sends a message $\mathtt{FastAck}(b_0, \mathtt{id}(c), D, P)$ to all replicas. Choose some replica $q \neq p$ in $\mathcal{SQ}(b_0)$. Consider the point in time when q has already received c as well as the above message from p. This takes at most 2δ units of time. Because the system is stable, recovery never takes place after time t_0; hence $\mathtt{id}(c) \in \mathit{Preaccept}$. Furthermore, we know that $\mathsf{bal} = \mathsf{cbal} = b_0$ and $\mathsf{status} = \text{NORMAL}$ at q. Pick some command $d \in D$. From the fact that the system is stable, replica q has already received a message $\mathtt{FastAck}$ regarding command d from the leader. Hence, by a short induction, the precondition $D \subseteq \mathit{Accept} \cup \mathit{Commit}$ is eventually true at q. It follows that replica q eventually sends a message $\mathtt{SlowAck}(b_0, id(c))$ to $\mathtt{client}(c)$ (either at line 27, or at line 29). From what precedes, $\mathtt{client}(c)$ eventually receives a message $\mathtt{SlowAck}(b_0, id(c))$ from all such replicas. Consider the point in time at which this holds for some slow quorum of the ballot b_0. Eventually, the client also receives a $\mathtt{Reply}$ message from the leader. When these two happen, the precondition at line 4 is true. In this case, the response of command c is known at the client after 3δ units of time. □

B CURP for Geo-Replication

Although CURP is a primary-backup protocol, its authors also describe a variation for eventually synchronous systems [41, Appendix B.2]. This variation reuses a black-box leader-driven SMR protocol to enforce the agreement on the state-machine transitions at the replicas. In Figures 14 and 15 we use N^{2}Paxos for this purpose. For the sake of brevity, we omit recovery from the protocol's description. CURP+N^{2}Paxos slightly differs from the logic described in the CURP paper [41], as highlighted in blue in the figures.

CURP+N^{2}Paxos is a leader-based protocol. The non-leader replicas (aka followers) acknowledge each transition proposed by the leader. The protocol uses a set of witnesses, which are co-located with the followers and durably store each state-machine command (variable Unsynced in Figure 15). If a command is conflict-free (line 21), its client receives the result via the fast path after a single round-trip to the replicas (line 5). Otherwise, the command takes the slow path (line 7).

```
1 func submit($c$):
2     send Record($c$) to $P$

3 when received Reply($b, id, r$) from leader($b$)
  and RecordAck($b, id$) from all $q \in Q$
4     pre: $|Q| > 3N/4$
5     response($id, r$)

6 when received SyncReply($id, r$)
7     response($id, r$)
```

Figure 14: CURP+N^{2}Paxos: client.

CURP+N^{2}Paxos cuts one message delay in comparison to a combination of CURP with Paxos or Raft. This is because a replica sends an `AcceptAck` message to all the replicas and not simply to the leader (line 27). Each replica can now rapidly find out when a command is committed, and the replica closest to the client returns the response (lines 36-37). This change is helpful in geo-distributed systems, where reducing the latency is key for performance. Another difference with CURP [41, Appendix B.2] is the client's logic. To address the problem of zombie leaders, the authors of CURP propose to rely on a cached value of the term number (or ballot) at each client. Instead, we piggyback this value on the messages addressed to the client, which simplifies the protocol (see line 3 in Figure 14).

```
8  when received Record($c$) from client $q$
9      pre: $p = \text{leader}(b) \land \text{cmd}[\text{id}(c)] = \bot$
10     $\text{cmd}[\text{id}(c)] \leftarrow c$
11     $\text{next} \leftarrow \text{next} + 1$
12     $\text{log}[\text{next}] \leftarrow id$
13     $\text{pending\_log} \leftarrow \text{pending\_log} \cdot c$
14     if $\forall n < \text{next}. \text{cmd}[\text{log}[n]] \bowtie c \implies$
         $\text{log}[n] \in \text{Commit}$ then
15         let $r = \text{opt\_exec}(\text{pending\_log}, \text{state})$
16         send Reply($\text{bal}, \text{id}(c), r$) to $q$
17     send Accept($\text{bal}, \text{id}(c), \text{next}$) to all

18 when received Record($c$) from client $q$
19     pre: $p \neq \text{leader}(b) \land \text{cmd}[\text{id}(c)] = \bot$
20     $\text{cmd}[\text{id}(c)] \leftarrow c$
21     if $\forall id \in \text{Unsynced}. \neg(\text{cmd}[id] \bowtie c)$ then
22         $\text{Unsynced} \leftarrow \text{Unsynced} \cup \{\text{id}(c)\}$
23         send RecordAck($\text{bal}, \text{id}(c)$) to $q$

24 when received Accept($b, id, n$) from $q$
25     pre: $\text{bal} = b \land \text{cmd}[id] \neq \bot$
26     $\text{log}[n] \leftarrow id$
27     send AcceptAck($b, id, n$) to all

28 when received AcceptAck($b, id, n$) from all $q \in Q$
29     pre: $\text{bal} = b \land |Q| > N/2 \land \text{cmd}[id] \neq \bot \land$
         $\text{log}[n] = id \land \forall n' < n. \text{log}[n'] \in \text{Commit}$
30     $\text{Commit} \leftarrow \text{Commit} \cup \{id\}$
31     if $p \neq \text{leader}(b)$ then $\text{Unsynced} \leftarrow \text{Unsynced} \setminus \{id\}$

32 when there exists $(n, id)$ such that $id \notin \text{Exec} \land$
   $\text{log}[n] = id \land id \in \text{Commit} \land \forall n' < n. \text{log}[n'] \in \text{Exec}$
33     $\text{Exec} \leftarrow \text{Exec} \cup \{id\}$
34     let $(r, \text{state}) = \text{exec}(\text{cmd}[id], \text{state})$
35     if $p = \text{leader}(\text{bal})$ then remove($\text{pending\_log}, id$)
36     if $IsClosest(p, \text{client}(id))$ then
37         send SyncReply($id, r$)
```

Figure 15: CURP+N^{2}Paxos: replica p.

C Configuration of the Experiments from §5

	ap-south-1	ap-northeast-1	eu-west-3	us-west-1	af-south-1	ap-east-1	ap-southeast-2	ca-central-1	eu-west-1	sa-east-1	us-east-1	us-east-2
ap-northeast-1	128 ms											
eu-west-3	108 ms	217 ms										
us-west-1	231 ms	110 ms	143 ms									
af-south-1	164 ms	359 ms	152 ms	292 ms								
ap-east-1	91 ms	54 ms	201 ms	156 ms	254 ms							
ap-southeast-2	152 ms	111 ms	281 ms	140 ms	414 ms	130 ms						
ca-central-1	191 ms	146 ms	86 ms	81 ms	228 ms	192 ms	200 ms					
eu-west-1	125 ms	203 ms	20 ms	130 ms	163 ms	216 ms	258 ms	72 ms				
sa-east-1	301 ms	259 ms	197 ms	175 ms	344 ms	304 ms	314 ms	127 ms	180 ms			
us-east-1	190 ms	147 ms	84 ms	64 ms	232 ms	193 ms	201 ms	17 ms	71 ms	116 ms		
us-east-2	201 ms	136 ms	93 ms	55 ms	242 ms	180 ms	191 ms	27 ms	81 ms	125 ms	18 ms	
us-west-2	221 ms	99 ms	135 ms	24 ms	277 ms	146 ms	142 ms	61 ms	121 ms	176 ms	65 ms	54 ms

Figure 16: Latency table of AWS regions.

Replicas
ap-south-1
ap-northeast-1
eu-west-3
us-west-1
af-south-1

Clients
ap-east-1
ap-northeast-1
ap-southeast-2
eu-west-1
ca-central-1
sa-east-1
us-east-1
us-east-2
us-west-1
us-west-2

Figure 17: Regions used for replicas and clients.