State-Machine Replication for Planet-Scale Systems

Vitor Enes
INESC TEC and University of Minho

Carlos Baquero
INESC TEC and University of Minho

Tuanir França Rezende
Télécom SudParis

Alexey Gotsman
IMDEA Software Institute

Matthieu Perrin
University of Nantes

Pierre Sutra
Télécom SudParis

Abstract

Online applications now routinely replicate their data at multiple sites around the world. In this paper we present Atlas, the first state-machine replication protocol tailored for such planet-scale systems. Atlas does not rely on a distinguished leader, so clients enjoy the same quality of service independently of their geographical locations. Furthermore, client-perceived latency improves as we add sites closer to clients. To achieve this, Atlas minimizes the size of its quorums using an observation that concurrent data center failures are rare. It also processes a high percentage of accesses in a single round trip, even when these conflict. We experimentally demonstrate that Atlas consistently outperforms state-of-the-art protocols in planet-scale scenarios. In particular, Atlas is up to two times faster than Flexible Paxos with identical failure assumptions, and more than doubles the performance of Egalitarian Paxos in the YCSB benchmark.

CCS Concepts: • **Theory of computation → Distributed algorithms**.

Keywords: Fault tolerance, Consensus, Geo-replication.

ACM Reference Format:
Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. 2020. State-Machine Replication for Planet-Scale Systems. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3342195.3387543

1 Introduction

Modern online applications run at multiple sites scattered across the globe: they are now *planet-scale*. Deploying applications in this way enables high availability and low latency, by allowing clients to access the closest responsive site. A major challenge in developing planet-scale applications is that many of their underlying components, such as coordination kernels [5, 13] and critical databases [7], require strong guarantees about the consistency of replicated data.

The classical way of maintaining strong consistency in a distributed service is *state-machine replication (SMR)* [30]. In SMR, a service is defined by a deterministic state machine, and each site maintains its own local 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* [11], providing an illusion that each command executes instantaneously throughout the system.

Unfortunately, existing SMR protocols are poorly suited to planet-scale systems. Common SMR protocols, such as Paxos [15] and Raft [27], are rooted in cluster computing where a *leader* site determines the ordering of commands. This is unfair to clients far away from the leader. It impairs scalability, since the leader cannot be easily parallelized and thus becomes a bottleneck when the load increases. It also harms availability as, if the leader fails, the system cannot serve requests until a new one is elected. Moreover, adding more sites to the system does not help, but on the contrary, hinders performance, requiring the leader to replicate commands to more sites on the critical path. This is a pity, as geo-replication has a lot of potential for improving performance, since adding sites brings the service closer to clients.

To fully exploit the potential of geo-replication, we propose Atlas, a new SMR protocol tailored to planet-scale systems with many sites spread across the world. In particular, Atlas improves client-perceived latency as we add sites closer to clients. The key to the Atlas design is an observation that common SMR protocols provide a level of fault-tolerance that is unnecessarily high in a geo-distributed setting. These protocols allow any minority of sites to fail simultaneously: e.g., running a typical protocol over 13 data centers would tolerate 6 of them failing. However, natural disasters leading to the loss of a data center are rare, and planned downtime can be handled by reconfiguring the unavailable site out of the system [15, 31]. Furthermore, temporary data center outages (e.g., due to connectivity issues) typically have a short duration [20], and, as we confirm experimentally in §5, rarely happen concurrently. For this reason, industry practitioners assume that the number of concurrent site failures in a geo-distributed system is low, e.g. 1 or 2 [7]. Motivated by this, our SMR protocol allows choosing the

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387543

maximum number of sites that can fail (f) independently of the overall number of sites (n), and is optimized for small values of the former. Our protocol thus trades off higher fault tolerance for higher scalability[1].

In more detail, like previously proposed protocols such as Egalitarian Paxos (EPaxos) [24] and Mencius [21], our protocol is *leaderless*, i.e., it orders commands in a decentralized way, without relying on a distinguished leader site. This improves availability and allows serving clients with the same quality of service independently of their geographical locations. As is common, our protocol also exploits the fact that commands in SMR applications frequently commute [5, 7], and for the replicated machine to be linearizable, it is enough that replicas only agree on the order of non-commuting commands [16, 28]. This permits processing a command in one round trip from the closest replica using a *fast path*, e.g., when the command commutes with all commands concurrently submitted for execution. In the presence of concurrent non-commuting commands, the protocol may sometimes have to take a *slow path*, which requires two round trips.

Making our protocol offer better latency for larger-scale deployments required two key innovations in the baseline scheme of a leaderless SMR protocol. First, the lower latency of the fast path in existing protocols comes with a downside: the fast path must involve a *fast quorum* of replicas bigger than a majority, which increases latency due to accesses to far-away replicas. For example, in Generalized Paxos [16] the fast quorum consists of at least $\frac{2n}{3}$ replicas, and in EPaxos of at least $\frac{3n}{4}$ replicas. To solve this problem, in Atlas the size of the fast quorum is a function of the number of allowed failures f – namely, $\lfloor \frac{n}{2} \rfloor + f$. Smaller values of f result in smaller fast quorums, thereby decreasing latency. Furthermore, violating the assumption the protocol makes about the number of failures may only compromise liveness, but never safety. In particular, if more than f transient outages occur, due to, e.g., connectivity problems, Atlas will just block until enough sites are reachable.

A second novel feature of Atlas is that it can take the fast path even when non-commuting commands are submitted concurrently, something that is not allowed by existing SMR protocols [16, 24]. This permits processing most commands via the fast path when the conflict rate is low-to-moderate, as is typical for SMR applications [5, 7]. Moreover, when $f = 1$ our protocol *always* takes the fast path and its fast quorum is a plain majority.

The biggest challenge we faced in achieving the above features – smaller fast quorums and a flexible fast-path condition – was in designing a correct failure recovery mechanism for Atlas. Failure recovery is the most subtle part of a SMR protocol with a fast path because the protocol needs to recover the decisions reached by the failed replicas while they were short-cutting some of the protocols steps in the fast path. This is only made more difficult with smaller fast quorums, as a failed process leaves information about its computations at fewer replicas. Atlas achieves its performant fast path while having a recovery protocol that is significantly simpler than that of previous leaderless protocols [2, 24] and has been rigorously proved correct.

As an additional optimization, Atlas also includes a novel mechanism to accelerate the execution of linearizable reads and reduce their impact on the protocol stack. This improves performance in read-dominated workloads.

We experimentally evaluate Atlas on Google Cloud Platform using 3 to 13 sites spread around the world. As new replicas are added closer to clients, Atlas gets faster: going from 3 to 13 sites, the client-perceived latency is almost cut by half. We also experimentally compare Atlas with Flexible Paxos [12] (a variant of Paxos that also allows selecting f independently of n), EPaxos and Mencius. Atlas consistently outperforms these protocols in planet-scale scenarios. In particular, our protocol is up to two times faster than Flexible Paxos with identical failure assumptions ($f = 1, 2$), and more than doubles the performance of EPaxos in mixed YCSB workloads [6].

2 State-Machine Replication

We consider an asynchronous distributed system consisting of n processes $\mathcal{P} = \{1, \ldots, n\}$. At most f processes may fail by crashing (where $1 \leq f \leq \lfloor \frac{n-1}{2} \rfloor$), but processes do not behave maliciously. In a geo-distributed deployment, each process represents a data center, so that a failure corresponds to the outage of a whole data center. Failures of single machines are orthogonal to our concerns and can be masked by replicating a process within a data center using standard techniques [15, 27]. We call a majority of processes a *(majority) quorum*. We assume that the set of processes is static. Classical approaches can be used to add reconfiguration to our protocol [15, 24]. Reconfiguration can also be used in practice to allow processes that crash and recover to rejoin the system.

State-machine replication (SMR) is a common way of implementing strongly consistent replicated services [30]. A service is defined by a deterministic state machine with an appropriate set of *commands*, denoted by $\mathcal{C}$. Processes maintain their own local copy of the state machine, and proxy the access to the replicated service by client applications (not modeled). An *SMR protocol* coordinates the execution of commands at the processes, ensuring that service replicas stay in sync. The protocol provides a command `submit`(c), which allows a process to submit a command $c \in \mathcal{C}$ for execution on behalf of a client. The protocol may also trigger an event

[1] Apart from data centers being down, geo-distributed systems may also exhibit network partitionings, which partition off several data centers from the rest of the system. Our protocol may block for the duration of the partitioning, which is unavoidable due to the CAP theorem [10].

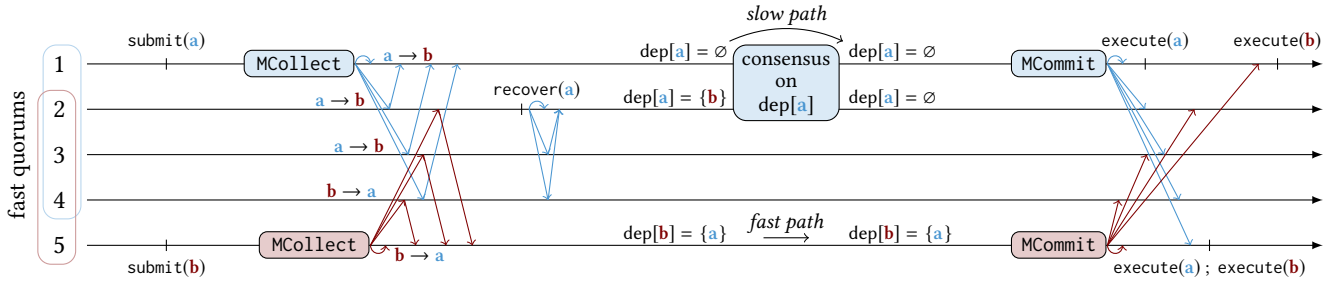

Figure 1. Example of processing two conflicting commands a and b in Atlas with $n = 5$ processes and up to $f = 2$ failures. We omit the messages implementing consensus and depict this step abstractly by the consensus box.

execute(c) at a process, asking it to apply c to the local service replica; after execution, the process that submitted the command may return the outcome of c to the client. Without loss of generality, we assume that each submitted command is unique.

The strongest property a replicated service implemented using SMR may satisfy is *linearizability* [11]. Informally, this means that commands appear 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. As observed in [16, 28], for the replicated service to be linearizable, the SMR protocol does not need to ensure that commands are executed at processes in the exact same order: it is enough to agree on the order of non-commuting commands.

We now give the specification of the SMR protocol. We say that commands c and d *commute* if in 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 in s, and vice versa. If commands do not commute, we say that they *conflict*[2]. We write $c \mapsto_i d$ when c and d conflict and process $i \in \mathcal{P}$ executes c before executing d. We also define the following *real-time order*: $c \rightsquigarrow d$ if c was executed at some process before d was submitted. Let $\mapsto = \rightsquigarrow \cup (\bigcup_{i \in \mathcal{P}} \mapsto_i)$. Then, the specification of the SMR protocol is given by the following properties:

Validity. If a process executes a command c, then some process submitted c before.

Integrity. A process executes each command at most once.

Ordering. The relation $\mapsto$ is acyclic.

Note that the Ordering property enforces that conflicting commands are executed in a consistent manner across the system. In particular, it prevents two conflicting commands from being executed in contradictory orders by different processes. If the SMR protocol satisfies the above properties, then the replicated service implemented using it is linearizable (we prove this in [9, §B]). In the following sections we present Atlas, which satisfies the above specification.

[2]Detecting if two commands conflict must be possible without executing them. In practice, this information can often be extracted from the API provided by the replicated service. In cases when such inference is infeasible, it is always safe to consider that a pair of commands conflict.

3 The Atlas Protocol

To aid understanding, we first illustrate by example the message flow of the Atlas protocol (§3.1), which corresponds to a common structure of leaderless SMR protocols [24]. We then describe the protocol in detail (§3.2).

3.1 Overview

Figure 1 illustrates how Atlas processes two conflicting commands, a and b, with $n = 5$ processes and at most $f = 2$ failures. At a given process, a command usually goes through several *phases*: the initial phase start, then collect, commit and execute (an additional phase recover is used when handling failures).

Command a starts its journey when submit(a) is invoked at process 1. We call process 1 the initial *coordinator* of a. This coordinator is initial because, if it fails or is slow, another process may take over. Command a then enters the collect phase at process 1, whose goal is to compute the set of commands that are *dependencies* of a, denoted by dep[a]. These dependencies are later used to determine the order of execution of conflicting commands. To compute dependencies, process 1 sends an MCollect message containing command a to a *fast quorum* of processes, which is at least a majority but may be bigger. In our example the fast quorum picked by 1 is $\{1, 2, 3, 4\}$.

Each process in the fast quorum returns the set of commands conflicting with a that it received before a. In Figure 1, $\rightarrow$ indicates the order in which processes receive commands. For instance, process 4 receives b first, whereas the other fast-quorum processes do not receive any command before a. Based on the replies, process 1 computes the value of dep[a] (as described in the next section); in our example this happens to be $\emptyset$.

If a coordinator of a command is suspected to have failed, another process may try to take over. In Figure 1, process 2 suspects 1 and becomes another coordinator of a, denoted

by recover(a). Process 2 contacts a majority quorum of processes {2, 3, 4} and computes its own version of the dependencies of a: dep[a] = {b}.

Dependencies are used to determine the order in which conflicting commands are executed, and all processes have to execute conflicting commands in the same order. To ensure this, the coordinators of command a need to reach a consensus on the value of dep[a]. This is implemented using an optimized variant of single-decree Paxos [15], with all n processes acting as acceptors. In our example, this makes the processes agree on dep[a] = ∅. The use of consensus represents the *slow path* of the protocol.

If a coordinator can ensure that all the values that can possibly be proposed to consensus are the same, then it can take the *fast path* of the protocol, avoiding the use of consensus. In Figure 1, this is the case for process 5 coordinating command b. For a process to take the fast path, we require it to receive a response from every process in the fast quorum, motivating the name of the latter.

After consensus or the shortcut via the fast path, a coordinator of a command sends its final dependencies to other processes in an MCommit message. A process stores these dependencies and marks the command as having entered the COMMIT phase. A command can be executed (and thereby transition to the EXECUTE phase) only after all its dependencies are in the COMMIT or EXECUTE phases. Since in our example dep[a] = ∅, processes can execute command a right after receiving its final dependencies (∅). This is exploited by processes 1 and 2 in Figure 1. However, as dep[b] = {a}, processes must delay the execution of b until a is executed. This is the case for processes 3, 4 and 5 in Figure 1. Such an execution mechanism guarantees that the conflicting commands a and b are executed in the same order at all processes.

3.2 Protocol in Detail

Algorithm 1 specifies the ATLAS protocol at process $i \in \mathcal{P}$ in the failure-free case. We assume that self-addressed protocol messages are delivered immediately.

3.2.1 Start phase. A client submits a command $c \in \mathcal{C}$ by invoking submit(c) at one of the processes running ATLAS, which will serve as the initial command coordinator. When submit(c) is invoked at a process i (line 1), this coordinator first assigns to command c a unique identifier – a pair $\langle i, l \rangle$ where $l - 1$ is the number of commands submitted at process i before c. In the following we denote the set of all identifiers by $\mathcal{I}$. At the bottom of Algorithm 1, we summarize the data maintained by each process for a command with identifier $id \in \mathcal{I}$. In particular, the mapping cmd stores the payload of the command, and the mapping phase tracks the progress of the command through phases. For brevity, the name of the phase written in lower case also denotes all the identifiers in that phase, e.g., $start = \{id \in \mathcal{I} \mid \text{phase}[id] = \text{START}\}$.

Once the coordinator assigns an identifier to c, the command starts its COLLECT phase, whose goal is to compute a set of identifiers that are the *dependencies* of c. At the end of this phase, the coordinator sends an MCommit(id, c, D) message including the computed dependencies D. Before this, it agrees with other possible coordinators on the same final value of D, resulting in the following invariant.

INVARIANT 1. For any two messages MCommit(id, c, D) and MCommit(id, c', D') sent, $c = c'$ and $D = D'$.

Hence, each identifier is associated with a unique command and final set of dependencies. The key property of dependencies is that, for any two distinct conflicting commands, one has to be a dependency of the other. This is stated by the following invariant.

INVARIANT 2. Assume that messages MCommit(id, c, D) and MCommit(id', c', D') have been sent. If $id \neq id'$ and $conflict(c, c')$ then either $id' \in D$ or $id \in D'$, or both.

This invariant is key to ensure that conflicting commands are executed in the same order at all processes, since we allow processes to execute commands that are not a dependency of each other in any order. We next explain how ATLAS ensures the above invariants.

3.2.2 Collect phase. To compute the dependencies of a command c, its coordinator first computes the set of commands it knows about that conflict with c (denoted by *past*, line 3) using a function $\text{conflicts}(c) = \{id \notin start \mid conflict(c, \text{cmd}[id])\}$. The coordinator then picks a fast quorum Q of size $\lfloor \frac{n}{2} \rfloor + f$ that includes itself (line 4) and sends an MCollect message with the information it computed to all processes in Q.

Upon receiving an MCollect message from the coordinator, a process in the fast quorum computes its contribution to c's dependencies as the set of commands that conflict with c, combined with *past* (line 8). The process stores the computed dependencies, command c and the fast quorum Q in mappings dep, cmd and quorum, respectively, and sets the command's phase to COLLECT. The process then replies to the coordinator with an MCollectAck message, containing the computed dependencies (line 11).

Once the coordinator receives an MCollectAck message from all processes in the fast quorum (line 13), it computes the dependencies for the command as the union of all reported dependencies $D = \bigcup_Q dep = \bigcup\{dep_j \mid j \in Q\}$ (line 14). Since a fast quorum contains at least a majority of processes, the following property implies that this computation maintains Invariant 2.

PROPERTY 1. Assume two conflicting commands with identifiers id and id' and dependencies D and D' computed as in line 14 over majority quorums. Then either $id' \in D$ or $id \in D'$, or both.

Algorithm 1: Atlas protocol at process i: failure-free case.

```
1  function submit(c)
2    id ← ⟨i, min{l | ⟨i, l⟩ ∈ start}⟩
3    past ← conflicts(c)
4    Q ← fast_quorum(i)
5    send MCollect(id, c, past, Q) to Q

6  receive MCollect(id, c, past, Q) from j
7    pre: id ∈ start
8    dep[id] ← conflicts(c) ∪ past
9    cmd[id] ← c; quorum[id] ← Q
10   phase[id] ← COLLECT
11   send MCollectAck(id, dep[id]) to j

12 receive MCollectAck(id, dep_j) from all j ∈ Q
13   pre: id ∈ collect ∧ Q = quorum[id]
14   D ← ⋃_Q dep
15   if ⋃_Q dep = ⊎_Q dep then
16     send MCommit(id, cmd[id], D) to all
17   else
18     Q′ ← slow_quorum(i)
19     send MConsensus(id, cmd[id], D, i) to Q′

20 receive MConsensus(id, c, D, b) from j
21   pre: bal[id] ≤ b
22   cmd[id] ← c; dep[id] ← D
23   bal[id] ← b; abal[id] ← b
24   send MConsensusAck(id, b) to j

25 receive MConsensusAck(id, b) from Q
26   pre: bal[id] = b ∧ |Q| = f + 1
27   send MCommit(id, cmd[id], dep[id]) to all

28 receive MCommit(id, c, D)
29   pre: id ∉ commit ∪ execute
30   cmd[id] ← c; dep[id] ← D; phase[id] ← COMMIT
```

cmd[id] ← noOp	$\in \mathcal{C}$	Command
phase[id] ← START		Phase
dep[id] ← $\emptyset$	$\subseteq \mathcal{I}$	Dependency set
quorum[id] ← $\emptyset$	$\subseteq \mathcal{P}$	Fast quorum
bal[id] ← 0	$\in \mathbb{N}$	Current ballot
abal[id] ← 0	$\in \mathbb{N}$	Last accepted ballot

Proof. Assume that the property does not hold: there are two conflicting commands with distinct identifiers id and id' and dependencies D and D' such that $id' \notin D$ and $id \notin D'$. We know that D was computed over some majority Q and D' over some majority Q'. Since $id' \notin D$, we have: *(i)* the majority Q observed id before id'. Similarly, since $id \notin D'$: *(ii)* the majority Q' observed id' before id. However, as majorities Q and Q' must intersect, we cannot have both *(i)* and *(ii)*. This contradiction shows the required. □

For example, in Figure 1 coordinator 5 determines the dependencies for **b** using the computation at line 14 (coordinator 1 uses an optimized version of this computation presented in §4).

After computing the command's dependencies, its coordinator decides to either take the fast path (line 15) or the slow path (line 17). Both fast and slow paths end with the coordinator sending an MCommit message containing the command and its final dependencies.

3.2.3 Slow path. If the coordinator of a command is suspected to have failed, another process may try to take over its job and compute a different set of dependencies. Hence, before an MCommit message is sent, processes must reach an agreement on its contents to satisfy Invariant 1. They can always achieve this by running a consensus protocol – this is the slow path of Atlas. Consensus is implemented using single-decree (Flexible) Paxos [12]. For each identifier we allocate ballot numbers to processes round-robin, with ballot i reserved for the initial coordinator i and ballots higher than n for processes that try to take over. Every process stores for each identifier id the ballot number bal[id] it is currently participating in and the last ballot abal[id] in which it accepted a proposal (if any).

When the initial coordinator i decides to go onto the slow path, it performs an analog of Paxos Phase 2: it sends an MConsensus message with its proposal and ballot i to a *slow quorum* that includes itself (line 18)[3]. Following Flexible Paxos [12], the size of the slow quorum is only $f + 1$, rather than a majority like in classical Paxos. This minimizes the additional latency incurred on the slow path in exchange for using larger quorums in recovery (as described below). As usual in Paxos, a process accepts an MConsensus message only if its bal[id] is not greater than the ballot in the message (line 21). Then it stores the proposal, sets bal[id] and abal[id] to the ballot in the message, and replies to the coordinator with MConsensusAck. Once the coordinator gathers $f + 1$ such replies (line 26), it is sure that its proposal will survive the allowed number of failures f, and it thus broadcasts the proposal in an MCommit message (line 27).

3.2.4 Fast path. The initial coordinator of a command can avoid consensus when it can ensure that any process performing recovery will propose the same set of dependencies to consensus [17] – this is the fast path of Atlas, in which a command is committed after a single round trip to the closest fast quorum (line 16). In order to take the fast path, previous SMR protocols, such as Generalized Paxos [16] and EPaxos [24], require fast-quorum replies to match exactly.

[3]The initial coordinator i can safely skip Paxos Phase 1: since processes perform recovery with ballots higher than n, no proposal with a ballot lower than i can ever be accepted.

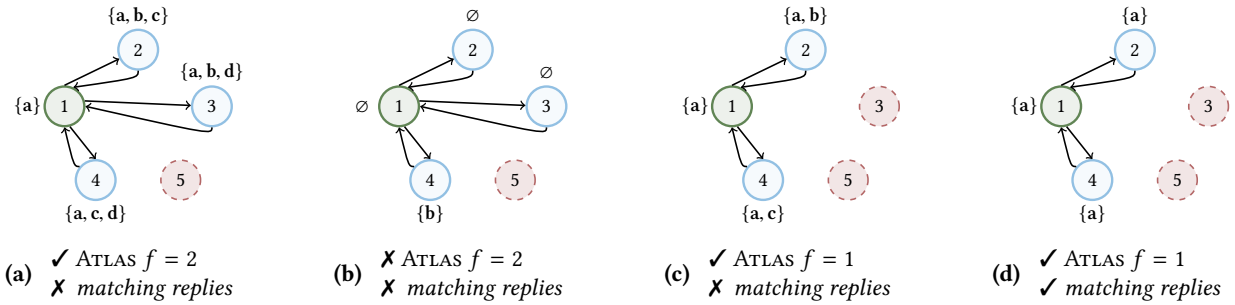

Figure 2. Examples in which the fast path is taken ✓ or not ✗, for both Atlas and protocols that require *matching replies* from fast-quorum processes, such as EPaxos [24]. All examples consider $n = 5$ processes while tolerating f faults. The coordinator is always process 1, and circles with a solid line represent the processes that are part of the fast quorum. Next to each process we depict the set of dependencies sent to the coordinator (e.g. $\{\mathbf{a}, \mathbf{b}\}$).

One of the key innovations of Atlas is that it is able to take the fast path even if this is not the case, e.g., when conflicting commands are submitted concurrently. This feature significantly improves performance in practice (§5).

In more detail, the coordinator takes the fast path if every dependency reported by some fast-quorum process is actually reported by at least f such processes. This is expressed by the condition $\bigcup_Q dep = \lfloor f \rfloor_Q dep$ in line 15, where

$$\lfloor f \rfloor_Q dep = \{id \mid \mathsf{count}(id) \geq f\};$$
$$\mathsf{count}(id) = |\{j \in Q \mid id \in dep_j\}|.$$

Figure 2 contains several examples that illustrate the flexibility of the above fast-path condition. All examples consider $n = 5$ processes while tolerating varying numbers of faults f. The example in Figure 2a considers Atlas $f = 2$. The coordinator of some command, process 1, picks a fast quorum $Q = \{1, 2, 3, 4\}$ of size $\lfloor \frac{n}{2} \rfloor + f = 4$. It receives replies $dep_1 = \{\mathbf{a}\}$, $dep_2 = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$, $dep_3 = \{\mathbf{a}, \mathbf{b}, \mathbf{d}\}$, $dep_4 = \{\mathbf{a}, \mathbf{c}, \mathbf{d}\}$. The coordinator then computes $\lfloor 2 \rfloor_Q dep = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$, i.e., all the dependencies reported at least twice. Since $\bigcup_Q dep = \lfloor 2 \rfloor_Q dep$, the coordinator takes the fast path. This is not the case for the example in Figure 2b where $\bigcup_Q dep = \{\mathbf{b}\} \neq \emptyset = \lfloor 2 \rfloor_Q dep$ ($\mathbf{b}$ is excluded from $\lfloor 2 \rfloor_Q dep$ because $\mathsf{count}(\mathbf{b}) = 1$). In this case the coordinator has to take the slow path. Back in Figure 1 we had the same situation: coordinator 1 had to take the slow path because dependency **b** was declared solely by process 4. On the other hand, coordinator 5 was able to take the fast path, because dependency a was declared by $f = 2$ processes: 2 and 3.

Notice that in Figure 2a, the coordinator takes the fast path even though dependencies reported by processes do not match, a situation which may arise when conflicting commands are submitted concurrently. Furthermore, when $f = 1$ we have $\{id \mid \mathsf{count}(id) < f\} = \emptyset$, so that the fast-path condition in line 15 always holds. Hence, Atlas $f = 1$ *always* takes the fast path, as is the case in Figures 2c and 2d. In contrast, EPaxos is able to take the fast path only in Figure 2d, since it is the only example in which fast-quorum replies match.

3.2.5 Recovery idea. The initial coordinator of a command may fail or be slow to respond, in which case Atlas allows a process to take over its role and recover the command and its dependencies. We start by describing the idea of the most subtle part of this mechanism – recovering decisions reached by failed coordinators via the fast path.

Let $D = \bigcup_Q dep = \lfloor f \rfloor_Q dep$ be some fast-path proposal (line 16). By definition of $\lfloor f \rfloor_Q dep$, each $id \in D$ was reported in the `MCollectAck` message of at least f fast-quorum processes. It follows that D can be obtained without $f - 1$ of those processes by taking the union of the dependencies reported by the remaining processes. Moreover, as the initial coordinator is always part of the fast quorum and each process in the quorum combines its dependencies with the ones declared by the coordinator (i.e., *past* in line 8), the latter is also not necessary to obtain D. Thus, the proposal D can be obtained without f fast-quorum processes including the initial coordinator (e.g., if the processes fail), by combining the dependencies reported by the remaining $\lfloor \frac{n}{2} \rfloor + f - f = \lfloor \frac{n}{2} \rfloor$ processes. The following property captures this observation.

Property 2. Any fast-path proposal can be obtained by taking the union of the dependencies sent in `MCollectAck` by at least $\lfloor \frac{n}{2} \rfloor$ fast-quorum processes that are not the initial coordinator.

As an example, assume that after the fast path is taken in Figure 2a, $f = 2$ processes inside the fast quorum fail, one of them being the coordinator, process 1. Independently of which $\lfloor \frac{n}{2} \rfloor = 2$ fast-quorum processes survive, the proposal is always recovered by set union: $\bigcup_{\{2,3\}} dep = \bigcup_{\{2,4\}} dep = \bigcup_{\{3,4\}} dep = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$.

Algorithm 2: Atlas protocol at process i: recovery.

```
31 function recover(id)
32   b ← i + n(⌊bal[id]/n⌋ + 1)
33   send MRec(id, cmd[id], b) to all

34 receive MRec(id, _, _) from j
35   pre: id ∈ commit ∪ execute
36   send MCommit(id, cmd[id], dep[id]) to j

37 receive MRec(id, c, b) from j
38   pre: bal[id] < b ∧ id ∉ commit ∪ execute
39   if bal[id] = 0 ∧ id ∈ start then
40     dep[id] ← conflicts(c); cmd[id] ← c
41   bal[id] ← b
42   phase[id] ← RECOVER
43   send MRecAck(id, cmd[id], dep[id], quorum[id],
                  abal[id], b) to j

44 receive MRecAck(id, cmd_j, dep_j, Q_j^0, ab_j, b) from all j∈Q
45   pre: bal[id] = b ∧ |Q| = n − f
46   if ∃k ∈ Q. ab_k ≠ 0 then
47     let k be such that ab_k is maximal
48     send MConsensus(id, cmd_k, dep_k, b) to all
49   else if ∃k ∈ Q. Q_k^0 ≠ ∅ then
50     Q' ← if id.1 ∈ Q then Q else Q ∩ Q_k^0
51     send MConsensus(id, cmd_k, ⋃_{Q'} dep, b) to all
52   else send MConsensus(id, noOp, ∅, b) to all
```

In the case of Figure 2b it is unsafe to take the fast path since the proposal may not be recoverable: the failure of process 4 would lead to losing the dependency **b**, since this dependency was reported exclusively by this process.

3.2.6 Recovery in detail. A process takes over as the coordinator for some command with identifier *id* by calling `recover(id)` (line 31 in Algorithm 2). In order to find out if a decision on the dependencies of *id* has been reached in consensus, the new coordinator first performs an analog of Paxos Phase 1. It picks a ballot number it owns higher than any it participated in so far (line 32) and sends an `MRec` message with this ballot to all processes.

Upon the receipt of such a message, in case *id* is already committed or executed (line 35), the process notifies the new coordinator with an `MCommit` message. Otherwise, as is standard in Paxos, the process accepts the `MRec` message only if the ballot in the message is greater than its bal[*id*] (line 38). In this case, if the process is seeing *id* for the first time (line 39), it computes its contribution to *id*'s dependencies as the set of conflicting commands (line 40). Then, the process sets bal[*id*] to the new ballot and phase[*id*] to RECOVER. Finally, the process replies with an `MRecAck` message containing all the information it has regarding *id*: the corresponding command (cmd), its current set of dependencies (dep), the ballot at which these were previously accepted (abal), and the fast quorum (quorum). Note that quorum[*id*] = ∅ if the process did not see the initial `MCollect` message, and abal[*id*] = 0 if the process has not yet accepted any consensus proposal.

In the `MRecAck` handler (line 44), the new coordinator computes its proposal given the information provided by processes and sends this proposal in an `MConsensus` message to all processes. As in Flexible Paxos, the new coordinator waits for $n - f$ `MRecAck` messages. This guarantees that, if a quorum of $f + 1$ processes accepted an `MConsensus` message with a proposal (which could have thus been sent in an `MCommit` message), the new coordinator will find out about this proposal. To maintain Invariant 1, if any process previously accepted a consensus proposal (line 46), by the standard Paxos rules [12, 15], the coordinator selects the proposal accepted at the highest ballot (line 47).

If no consensus proposal has been accepted before, the new coordinator checks whether any of the processes that replied has seen the initial `MCollect` message, by looking for any non-empty fast quorum (line 49). If the fast quorum is known, depending on whether the initial coordinator replied or not, there are two possible cases that we describe next.

1) The initial coordinator replies to the new one ($id.1 \in Q$, line 50). In this case the initial coordinator has not taken the fast path before receiving the `MRec` message from the new one, as it would have replied with `MCommit` instead of `MRecAck` (line 36). It will also not take the fast path in the future, since when processing the `MRec` message it sets the command phase to RECOVER (line 42), which invalidates the `MCollectAck` precondition (line 13). Since the initial coordinator never takes the fast path, the new coordinator can choose the command's dependencies in any way, as long as it maintains Invariant 2. By Property 1, this is satisfied if the coordinator chooses the set union of the dependencies declared by at least a majority of processes. Hence, the new coordinator takes the union of the dependencies reported by the $n - f \geq n - \lfloor \frac{n-1}{2} \rfloor \geq \lfloor \frac{n}{2} \rfloor + 1$ processes in Q (line 51).

2) The initial coordinator does not reply to the new one ($id.1 \notin Q$, line 50). In this case the initial coordinator could have taken the fast path and, if it did, the new coordinator must propose the same dependencies. Given that the recovery quorum Q has size $n - f$ and the fast quorum Q_k^0 has size $\lfloor \frac{n}{2} \rfloor + f$, the set of processes $Q' = Q \cap Q_k^0$ (line 51) contains at least $\lfloor \frac{n}{2} \rfloor$ fast-quorum processes (distinct from the initial coordinator, as it did not reply). Furthermore, recall that when a process from Q' replies to the new coordinator, it sets the command phase to RECOVER (line 42), which invalidates the `MCollect` precondition (line 7). Hence, if the initial coordinator took the fast path, then each process in Q' must have processed its `MCollect` before the `MRec` of the new coordinator, and reported in the latter the dependencies from the former. Then using Property 2, the new coordinator recovers the fast-path proposal by taking the union of the

dependencies from the processes in Q' (line 51). It can be shown that, even if the initial coordinator did not take the fast path, this computation maintains Invariant 2, despite Q' containing only $\lfloor \frac{n}{2} \rfloor$ processes and Property 1 requiring a majority of them. This is for the same reason this number of processes is sufficient in Property 2: dependencies declared by the initial coordinator are included into those declared by other fast-quorum processes (line 8).

It remains to address the case in which the process performing the recovery observes that no process saw the initial fast quorum, and consequently the submitted command (line 52). For instance, suppose that process i sends an $\texttt{MCollect}(id, c, \_, \_)$ only to process j and then fails. Further, assume that j receives another $\texttt{MCollect}(\_, c', \_, \_)$ from process k, replies with a dependency set that includes the identifier id of c, and also fails. Now, process k cannot execute c' without executing c (since c is a dependency of c'), and it cannot execute c because its payload has been lost. We solve this issue similarly to EPaxos: if a process takes over as the new coordinator and cannot find the associated payload, it may replace it by a special noOp command (line 52) that is not executed by the protocol and conflicts with all commands. With this, the final command for some identifier can take two possible values: the one submitted (line 1) or noOp. It is due to this that we include the command payload in addition to its dependencies into consensus messages associated with a given identifier (e.g., line 19), thus ensuring that a unique payload will be chosen (Invariant 1). Due to the possible replacement of a command by a noOp, the protocol actually ensures the following weakening of Invariant 2, which is still sufficient to establish its correctness.

Invariant 2′. Assume that messages $\texttt{MCommit}(id, c, D)$ and $\texttt{MCommit}(id', c', D')$ have been sent. If $id \neq id'$, $\mathit{conflict}(c, c')$, $c \neq \text{noOp}$ and $c' \neq \text{noOp}$, then either $id' \in D$ or $id \in D'$, or both.

3.2.7 Command execution. Algorithm 3 describes a background task employed by Atlas that is responsible for executing commands after they are committed. This task runs in an infinite loop trying to execute a *batch* of commands. We define a batch as the smallest set of committed identifiers $S \subseteq \mathit{commit}$ such that, for each identifier $id \in S$, its dependencies are in the batch or already executed: $\text{dep}[id] \subseteq S \cup \mathit{execute}$ (line 54). This ensures that a command can only be executed after its dependencies or in the same batch with them, which yields the following invariant.

Invariant 3. Assume two commands c and c' with identifiers id and id', respectively. If a process executes a batch of commands containing c before executing a batch containing c', then $id' \notin \text{dep}[id]$.

As processes agree on the dependencies of each command (Invariant 1), the batch in which a command is executed is equal in every process, as reflected in following invariant.

Algorithm 3: Atlas protocol: command execution.

```
53 loop
54   let S be the smallest subset of commit such that
       ∀id ∈ S. (dep[id] ⊆ S ∪ execute)
55   for id ∈ S ordered by < do
56     execute(cmd[id])
57     phase[id] ← EXECUTE
```

Invariant 4. If a process executes command c in batch S and another process executes the same command c in batch S', then $S = S'$.

Inside a batch, commands are ordered according to some fixed total order $<$ on identifiers (line 55). This guarantees that conflicting commands are executed in a consistent order across all processes.

Consider again the example in Figure 1, where the final dependencies are $\text{dep}[\mathbf{a}] = \emptyset$ and $\text{dep}[\mathbf{b}] = \{\mathbf{a}\}$. There are two cases, depending on the order in which processes commit the commands **a** and **b**:

- **a** then **b**: at processes 1 and 2. When the command **a** is committed, the processes execute it in a singleton batch, as it has no dependencies. When later the command **b** is committed, the processes execute it in a singleton batch too, since its only dependency **a** has already been executed.
- **b** then **a**: at processes 3, 4 and 5. When the command **b** is committed, the processes cannot execute it, as its dependency **a** has not yet been committed. When later the command **a** is committed, the processes execute two singleton batches: first **a**, then **b**.

Note that **a** is executed before **b** in both cases, thus ensuring a consistent execution order across processes.

Assume now we had final dependencies $\text{dep}[\mathbf{a}] = \{\mathbf{b}\}$ and $\text{dep}[\mathbf{b}] = \{\mathbf{a}\}$. In this case, independently of the order in which processes commit the commands, a batch will only be formed when both are committed. Since all processes will form the same batch containing both **a** and **b**, these commands will be executed in a predefined order on their identifiers, again ensuring a consistent execution order.

3.3 Atlas Properties and Comparison with EPaxos

Complexity. Atlas commits a command after two communication delays when taking the fast path, and four otherwise. As pointed out in §3.2, when $f = 1$, a fast quorum contains exactly a majority of processes and Atlas *always* takes the fast path. This is optimal for leaderless protocols [10, 18] and results in a significant performance pay-off (§5).

Fault tolerance. Atlas is parameterized by the number of tolerated concurrent faults f: smaller values of f yield smaller fast and slow quorums, thus reducing latency. As observed in the literature [7, 20] and as we experimentally

confirm in §5.1, assuming small values of f is acceptable for geo-distribution. Furthermore, violating our assumption that the number of failures is bounded by f may only compromise the liveness of the protocol, and never its safety: if more than f transient outages occur, due to, e.g., connectivity problems, Atlas will just block until enough sites are reachable.

Comparison with EPaxos. Atlas belongs to the family of leaderless SMR protocols. We now provide a concise comparison with the most prominent protocol in this family, EPaxos [24]. The two protocols share the message flow, including the splitting into fast and slow paths. However, as we demonstrate experimentally in §5, Atlas significantly outperforms EPaxos, which is due to a number of novel design decisions that we took.

First, EPaxos requires the conflicts reported by the fast quorum processes to the coordinator to match exactly, whereas Atlas allows processes to report different dependencies, as long as each dependency can be recovered after f failures. This allows Atlas to take the fast path even when non-commuting commands are submitted concurrently.

Second, Atlas allows choosing the number of failures f independently of the size of the system n, which yields fast quorums of size $\lfloor \frac{n}{2} \rfloor + f$. EPaxos assumes up to $\lfloor \frac{n}{2} \rfloor$ failures and sets the fast quorum size to $\lfloor \frac{3n}{4} \rfloor$. Our decision results in smaller quorums for small values of f, which are appropriate in planet-scale systems [7, 20, 22]; smaller quorums then result in lower latency. Note that EPaxos cannot be straightforwardly modified to exploit the independent bound on failures f due to its very complex recovery mechanism [23, 34] (which, in fact, has been recently shown to contain a bug [32]). In contrast, Atlas achieves its smaller fast quorums with a significantly simpler recovery protocol that is able to recover fast-path decisions using Property 2.

3.4 Correctness

We have rigorously proved Invariants 1 and 2 (see [9, §A]; we omit the easy proofs of Invariants 3 and 4). We now prove that the protocol invariants imply the correctness of Atlas, i.e., that it satisfies the SMR specification. The only nontrivial property is Ordering, which we prove next.

Lemma 1. The relation $\bigcup_{i=1}^{n} \mapsto_i$ is asymmetric.

Proof. By contradiction, assume that for some processes i and j and conflicting commands c and c' with identifiers id and id', we have $c \mapsto_i c'$ and $c' \mapsto_j c$; then $c \neq \text{noOp}$ and $c' \neq \text{noOp}$. By Integrity we must have $i \neq j$ and $c \neq c'$.

Assume first that c and c' are executed at process i in the same batch S. Then by Invariant 4 they also have to be executed at process j in the batch S. Since inside a batch commands are ordered using the fixed order $<$ on their identifiers, c and c' have to be executed in the same order at the two processes: a contradiction.

Assume now that c and c' are not executed at process i in the same batch. Then by Invariant 4 this also must be the case at process j. Hence, Invariant 3 implies that $id' \notin \text{dep}[id]$ at process i, and $id \notin \text{dep}[id']$ at process j. Then process i received $\texttt{MCommit}(id, c, D)$ with $id' \notin D$, and process j received $\texttt{MCommit}(id', c', D')$ with $id \notin D'$, which contradicts Invariant 2′. □

Lemma 2. Assume $c_1 \mapsto \ldots \mapsto c_n$ for $n \geq 2$. Whenever a process i executes c_n, some process has already executed c_1.

Proof. We prove the lemma by induction on n. The base case of $n = 2$ directly follows from the definition of $\mapsto$. Take $n > 3$ and assume $c_1 \mapsto \ldots \mapsto c_{n-1} \mapsto c_n$. Consider the moment when a process i executes c_n. We want to show that by this moment some process has already executed c_1. Since $c_{n-1} \mapsto c_n$, either $c_{n-1} \rightsquigarrow c_n$ or $c_{n-1} \mapsto_j c_n$ for some process j. Consider first the case when $c_{n-1} \rightsquigarrow c_n$. Then c_{n-1} is executed at some process k before c_n is submitted and, hence, before c_n is executed at process i. By induction hypothesis, c_1 is executed at some process before c_{n-1} is executed at process k and, hence, before c_n is executed at process i, as required. We now consider the case when $c_{n-1} \mapsto_j c_n$ for some process j. Since process i executes c_n, we must have either $c_{n-1} \mapsto_i c_n$ or $c_n \mapsto_i c_{n-1}$. The latter case would contradict Lemma 1, so that $c_{n-1} \mapsto_i c_n$. By induction hypothesis, c_1 is executed at some process before c_{n-1} is executed at process i and, hence, before c_n is executed at process i, as required. □

Proof of Ordering. By contradiction, assume that $c_1 \mapsto \ldots \mapsto c_n = c_1$ for $n \geq 2$. Then some process executed c_1. Consider the moment when the first process did so. By Lemma 2 some process has already executed c_1 before this, which yields a contradiction. □

4 Optimizations

This section presents two mechanisms employed by the Atlas protocol to accelerate command execution.

Reducing dependencies in the slow path. When computing dependencies in the slow path, instead of proposing $\bigcup_Q dep$ to consensus (line 19), the coordinator can propose $\biguplus_Q dep$. This allows Atlas to prune from dependencies those commands that been reported by less than f fast-quorum processes ($\{id \mid \text{count}(id) < f\}$) without breaking Invariant 2′. Smaller dependency sets allow batches to form more quickly in execution (Algorithm 3), thus reducing the delay between a command being committed and executed.

Back in Figure 1, command **b** was reported to coordinator 1 solely by process 4. Since **b** was reported by less than $f = 2$ processes, the above optimization allows coordinator 1 to prune **b** from the dependencies of **a**, thus proposing $\text{dep}[\mathbf{a}] = \emptyset$ to consensus. This maintains Invariant 2′, as **a** is still a dependency of **b**: $\text{dep}[\mathbf{b}] = \{\mathbf{a}\}$. In [9, §A.2.1] we prove that this optimization always maintains Invariant 2′.

Non-fault-tolerant reads. We observe that reads can be excluded from dependencies at lines 3 and 8 when the conflict relation between commands is transitive. In this case, a read is never a dependency and thus it will never block a later command, even if it is not fully executed, e.g., when its coordinator fails (or hangs). For this reason, reads can be executed in a non-fault-tolerant manner. More precisely, for some read with identifier id, the coordinator selects a plain majority as a fast quorum (line 4), independently of the value of f. Then, at the end of the COLLECT phase, it immediately commits id, setting dep[id] to the union of all dependencies returned by this quorum (line 16). This optimization, that we denote by NFR, accelerates the execution of linearizable reads and reduces their impact in the protocol stack. We show its correctness in [9, §B]. The transitivity requirement on conflicts is satisfied by many common applications. We experimentally evaluate the case of a key-value store in §5.

5 Performance Evaluation

In this section we experimentally compare ATLAS with Flexible Paxos (FPaxos) [12] and two leaderless protocols, EPaxos [24] and Mencius [21]. Mencius distributes the leader responsibilities round-robin among replicas; because of this, executing a command in Mencius requires contacting all replicas. As discussed previously, FPaxos uses a quorum of $f+1$ replicas in the failure-free case in exchange for a bigger quorum of $n-f$ replicas on recovery.

To improve the fairness of our comparison, ATLAS and EPaxos use the same codebase. This codebase consists of a server component, written in Erlang (3.7K SLOC), and a client component, written in Java (3.1K SLOC). The former commits commands, while the latter executes them. Thus, the implementation of two protocols differs only in the logic of the commit component. For Mencius and Paxos we use the Golang implementation provided by the authors of EPaxos [24], which we extended to support FPaxos.

Our evaluation takes place on Google Cloud Platform (GCP), in a federation of Kubernetes clusters [4]. The federation spans from 3 to 13 geographical regions spread across the world, which we call *sites*. When protocols are deployed in all 13 sites, we have 4 sites in Asia, 1 in Australia, 4 in Europe, 3 in North America, and 1 in South America. A site consists of a set of virtualized Linux nodes, each an 8-core Intel Xeon machine with 30 GB of memory (`n1-standard-8`). At a site, the SMR protocol and its clients execute on distinct machines. When benchmarking FPaxos, we take as leader the site that minimizes the standard deviation of clients-perceived latency. This site corresponds to the fairest location in the system, trying to satisfy uniformly all the clients.

5.1 Bounds on Failures

In a practical deployment of ATLAS, a critical parameter is the number of concurrent site failures f the protocol can tolerate. It has been reported that concurrent site failures are rare in geo-distributed systems [7]. However, the value of f should also account for asynchrony periods during which sites cannot communicate due to link failures: if more than f sites are unreachable in this way, our protocol may block for the duration of the outage. We have thus conducted an experiment to check that assuming small values of f is still appropriate when this is taken into account.

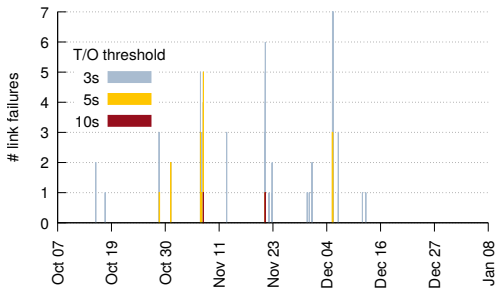

Figure 3. The number of simultaneous link failures among 17 sites in GCP when varying the timeout threshold.

Our experiment ran for 3 months (October 2018 – January 2019) among 17 sites, the maximal number of sites available in GCP at the time. During the experiment, sites ping each other every second (in the spirit of [20] but on a much larger scale). A link failure occurs between two sites when one of them does not receive a reply after a (tunable) amount of time. Figure 3 reports the number of simultaneous link failures for various timeout thresholds. Note that no actual machine crash occurred during the campaign of measurements.

When the timeout threshold is set to 10s, only two events occur, each with a single link failure. Fixing the threshold to either 3s or 5s leads to two events of noticeable length. During the first event, occurring on November 7, the links between the Canadian site (QC) and five others are slow for a couple of hours. During the second event, on December 8, the links between Taiwan (TW) and seven other sites are slow for around two minutes.

From the data collected, we compute the value of f as the smallest number of sites k such that, at any point in the experiment, crashing k sites would cover all the slow links. During our experiment, timeouts were reported on the links incident to at most a single site (e.g., the Canadian site on November 7). Thus, we may conclude that $f \leq 1$ held during the whole experiment, even with the smallest timeout threshold. In other words, ATLAS with $f \geq 1$ would have been always responsive during this 3-month experiment. In light of these results, we evaluate deployments of ATLAS in which f is set to 1, 2 or 3.

5.2 Benchmarks

Our first set of experiments uses a microbenchmark – a stub application that executes dummy commands (§5.2-§5.6). We

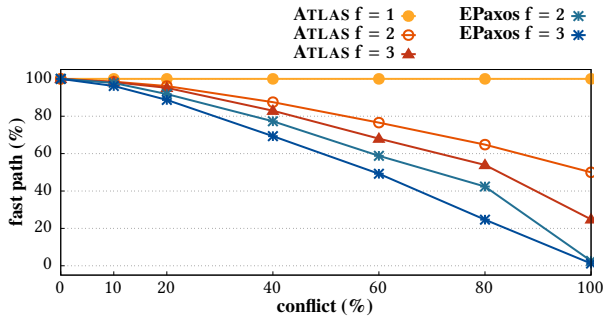

Figure 4. Ratio of fast paths for varying conflict rates.

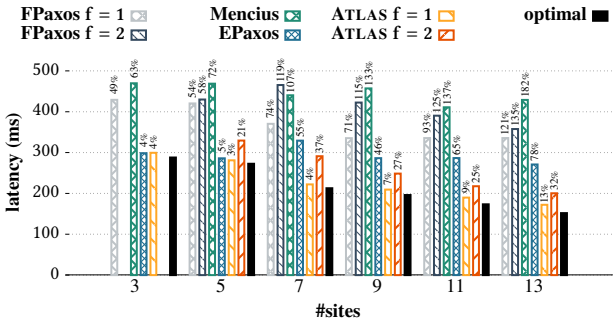

Figure 5. Latency when scaling-out from 3 to 13 sites with 1000 clients spread across 13 sites and 2% conflict rate. Percentages indicate the overhead wrt the optimal performance.

then evaluate Atlas with a geo-replicated key-value store under the YCSB workload [6] (§5.7). In our microbenchmark a varying number of closed-loop clients access the service at the closest (non-failed) site. Clients measure latency as the time between submitting a command and the system executing it. Each command carries a key of 8 bytes and (unless specified otherwise) a payload of 100 bytes. We assume that commands conflict when they carry the same key. To measure performance under a rate ρ of conflicting commands, a client chooses key 0 with a probability ρ, and some unique key otherwise.

5.3 Fast-Path Likelihood

Figure 4 evaluates the benefits of our new fast-path condition. To this end, it compares the fast-path ratio of Atlas and EPaxos for different conflict rates and values of f. The system consists of 3 sites when $f = 1$, 5 sites when $f = 2$, and 7 sites when $f = 3$. There is a single client per site (the results with more clients are almost identical).

As noted before, Atlas always commits a command on the fast path when $f = 1$. For higher values of f, our condition for taking the fast-path significantly improves its likelihood in comparison to EPaxos. With 5 sites and $f = 2$, when the conflict rate increases by 20%, the ratio of fast paths in EPaxos decreases on average by 20%. In contrast, the fast-path ratio in Atlas only decreases by 10%. When all commands conflict, EPaxos rarely takes the fast path, while Atlas does so for 50% of commands. Similar conclusions can be drawn from Figure 4 when the two protocols are deployed with $f = 3$.

5.4 Planet-Scale Performance

We now consider two planet-scale scenarios that motivate the design of Atlas. In these experiments we measure how the performance of Atlas evolves as the system scales up progressively from 3 to 13 sites. In the first experiment, the load on Atlas is constant, using a fixed number of clients spread across all 13 sites. We demonstrate how bringing the service closer to already existing clients, by adding new replicas, improves the latency these clients perceive. In the second experiment, each Atlas site hosts a fixed number of clients, so that the growth in the number of sites translates into increased load. This models a scenario where the service expands to new locations around the globe in order to serve new clients in these locations. In this case we demonstrate that Atlas gracefully copes with this growth, maintaining its performance in contrast to state-of-the-art SMR protocols.

Bringing the service closer to clients. We deploy 1000 clients equally spread over 13 sites, which issue commands at a fixed 2% conflict rate. Figure 5 reports how the average latency changes as Atlas is gradually deployed closer to client locations. The black bar in Figure 5 gives the average of the sum of round-trip latencies from a client to the closest coordinator, and from the coordinator to its closest majority. As clients execute no protocol logic and may not be co-located with sites, this gives the optimal latency for leaderless protocols (§3.3). The percentages on bars indicate the overhead of the different protocols with respect to this theoretical value.

As shown in Figure 5, Atlas improves its performance when deployed closer to clients: these can access a closer coordinator site, which in its turn accesses the closest fast quorum of sites. In Figure 5, the latency of Atlas $f = 1$ improves on average by 25ms whenever two new sites are added; for $f = 2$ this improvement is 33ms. For 13 sites, the optimal latency is 151ms, and Atlas $f = 1$ is only 13% above this value, with an average latency of 172ms; Atlas $f = 2$ is 32% above the optimum, with an average latency of 200ms. Overall, going from 3 to 13 sites with Atlas ($f = 1, 2$) cuts down client-perceived latency by 39%-42%.

As seen in Figure 5, the performance of Atlas greatly contrasts with that of the three other SMR protocols. With 13 sites, FPaxos executes commands after 336ms when $f = 1$ and after 358ms when $f = 2$, which is almost twice as slow as Atlas with identical failure assumptions. This large gap comes from the fact that, for a command to execute in a leader-based protocol, clients wait for four message delays on the critical path: a round trip from the client to the leader, and a round trip from the leader to a quorum.

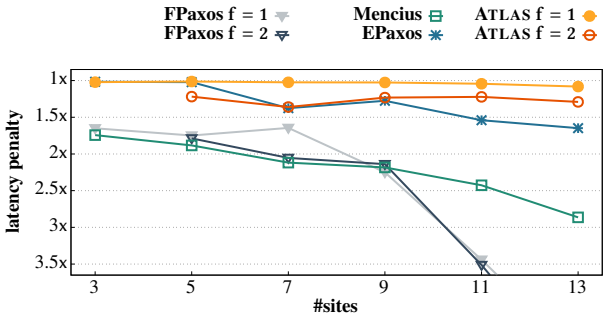

Figure 6. Latency penalty (with respect to the optimal performance) when scaling-out from 3 to 13 sites, with 128 clients deployed on each site, and 1% of conflict rate.

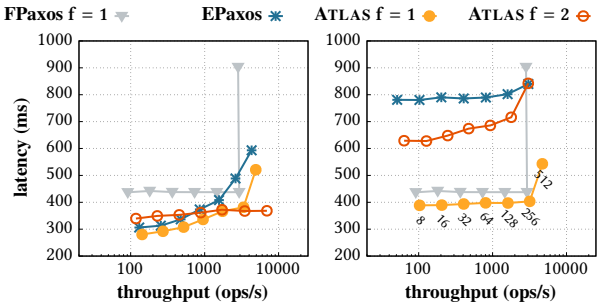

Figure 7. Throughput and latency with 5 sites when the load (number of clients) increases under moderate (left, 10%) and high (right, 100%) conflict rates.

The performance of EPaxos remains almost constant, within 10% of 300ms. With 13 sites, EPaxos is 78% slower than the optimum, and 57% slower than Atlas $f = 1$. This penalty is due to the large fast quorums it employs.

Finally, Mencius exhibits a high latency – above 400ms – in every configuration. This is because a replica needs to contact all the other replicas to execute a command, and thus, the performance of Mencius is bounded by the speed of its slowest replica.

Expanding the service. We now consider another planet-scale experiment that models a situation in which the service expands to new locations to serve new clients. The experiment considers 3 to 13 sites, with 128 clients per site, and each clients submits commands with a payload of 3KB. Figure 6 reports the latency penalty with respect to the optimal.

FPaxos $f = 1$ exhibits a latency penalty ranging from 1.7x to 4.7x (the last value is not shown in Figure 6 for readability). In particular, starting from 9 sites its performance degrades sharply with the increase in the number of sites and, hence, the number of clients. This happens because the leader cannot cope with the load, having to broadcast each command to all replicas. FPaxos $f = 2$ follows a similar trend.

EPaxos behaves better than FPaxos, hitting the optimal performance with 3 and 5 sites. However, starting from 11 sites, the latency of EPaxos becomes at best 50% of the optimum. Overall, due to its large fast quorums, the performance of EPaxos lowers as the number of sites increases.

In contrast to the prior two protocols, Atlas distributes the cost of broadcasting command payloads among replicas and uses small fast quorums. This allows the protocol to be within 4% of the optimum when $f = 1$, and within 26% when $f = 2$. Atlas is thus able to cope with the system growth without degrading performance.

5.5 Varying Load and Conflict Rate

To further understand the performance of Atlas, we conduct an experiment in which the load and the conflict rate vary. The protocol is deployed at 5 sites, and the load increases from 8 to 512 clients per site, under a moderate (10%) to high (100%) conflict rate. As before, messages carry a payload of 3KB. The results are presented in Figure 7, where we also compare with FPaxos $f = 1$ and EPaxos.

Under a 10% conflict rate (left-hand side of Figure 7) and with up to 64 clients per site, Atlas $f = 1$ executes commands with an average latency below 336ms. When this load doubles and then quadruples, the latency increases to respectively 366ms and 381ms. Compared to Atlas, the performance of EPaxos degrades faster with increasing load, yielding latencies of 368ms, 404ms and 484ms for 64, 128 and 256 clients per site, respectively. FPaxos performance is stable at 437ms for up to 256 clients per site, as the leader is capable of handling such a moderate load.

At a high load, with 512 clients per site, all the protocols but Atlas $f = 2$ saturate. In particular, FPaxos saturates because the leader is no longer capable of coping with the load. Although Atlas $f = 1$ is the most efficient protocol until saturation, its performance degrades sharply at 512 clients per site due to large batches formed during command execution. Interestingly, Atlas $f = 2$ behaves better due to the slow-path optimization in §4. Since this protocol uses larger fast quorums, the optimization allows it to prune dependencies that Atlas $f = 1$ cannot: while coordinators in Atlas $f = 1$ must include every dependency reported by a fast-quorum process for a given command, coordinators in Atlas $f = 2$ only include the dependencies reported by at least 2 fast-quorum processes. This reduces the size of batches in execution, improving the overall protocol latency.

With a 100% conflict rate (right-hand side of Figure 7), EPaxos performs worse than the remaining protocols. It executes commands with an average latency of at least 780ms, making the protocol unpractical in this context. As pointed out in §5.3, this is explained by its fast-path condition which rarely triggers when the conflict rate is high. In contrast, Atlas $f = 1$ is consistently the fastest protocol. Atlas is slower than FPaxos $f = 1$ only when providing a higher fault-tolerance level ($f = 2$).

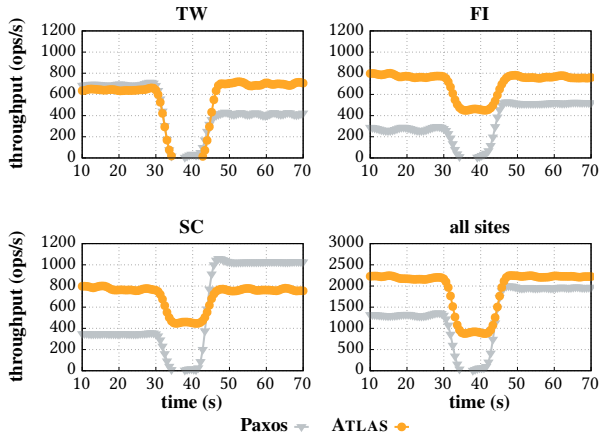

Figure 8. The impact of a failure on the throughput of Paxos and Atlas (3 sites, $f = 1$).

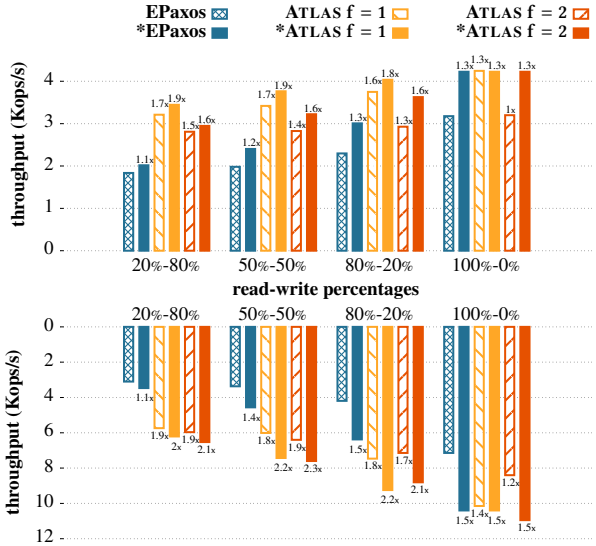

Figure 9. YCSB performance for update-heavy (20%-80%), balanced (50%-50%), read-heavy (80%-20%) and read-only (100%-0%) workloads, with 7 (top) and 13 sites (bottom). A * before the protocol name indicates that the NFR optimization (§4) is enabled. The number at the top of each bar indicates the speed-up over (vanilla) EPaxos.

5.6 Availability under Failures

Figure 8 depicts an experiment demonstrating that Atlas is inherently more available than a leader-driven protocol. The experiment runs across 3 sites: Taiwan (TW), Finland (FI) and South Carolina (SC). Such configuration tolerates a single site failure, so FPaxos is the same as Paxos. We do not evaluate EPaxos, as its availability guarantees are similar to those of Atlas in this configuration. Each site hosts 128 closed-loop clients. Half of the clients issue commands targeting key 0 and the other half issue commands targeting a unique key per client. Hence, commands by clients in the first half conflict with each other, while commands by clients in the second half commute with all commands by a different client.

After 30s of execution, the SMR service is abruptly halted at the TW site, where the Paxos leader is located. Based on the measurements reported in §5.1, we set the timeout after which a failure is suspected to 10s for both protocols. Upon detecting the failure, the clients located at the failed site (TW) reconnect to the closest alive site, SC. In the case of Paxos, the surviving sites initiate recovery and elect SC as the new leader. In the case of Atlas, the surviving sites recover the commands that were initially coordinated by TW.

As shown in Figure 8, Paxos blocks during the recovery time. In contrast, Atlas keeps executing commands, albeit at a reduced throughput. The drop in throughput happens largely because the clients issuing commands on key 0 (50% of all clients) collect as dependencies some of the commands being recovered (those that also access key 0). The execution of the former commands then blocks until the latter are recovered. In contrast, the clients at non-failed sites issuing commands with per-client keys continue to operate as normal. Since commands by these clients commute with those by other clients, their execution never blocks on the commands being recovered. This means that these clients operate without disruption during the whole experiment.

The bottom right plot contains the aggregate throughput of the system. Before failure, Atlas is almost two times faster than Paxos, and operates consistently better during the whole experiment. Note, however, that Paxos has a slightly higher throughput at the leader (TW) before the crash, and at the new leader (SC) after recovery. This is due to the delay between committing and executing commands in Atlas.

5.7 Key-Value Store Service

Our final experiment compares Atlas and EPaxos when the protocols are applied to a replicated key-value store (KVS) service. When accessing a KVS record stored under key k, a client executes either command $read(k)$ to fetch its content, or $write(k, v)$ to update it to value v. To benchmark the performance of the replicated KVS we use the Yahoo! Cloud Serving Benchmark (YCSB) [6]. We apply four types of workloads, each with a different mix of *read*/*write* operations. The KVS contains 10^6 records and all workloads select records following a Zipfian distribution with the default YCSB skew.

In this experiment, Atlas ($f = 1, 2$) and EPaxos are deployed over 7 and 13 sites (respectively, top and bottom of Figure 9). At each site running the benchmark we execute 128 YCSB client threads. The protocol name is preceded with the * symbol if the NFR optimization is enabled. As pointed out in §4, this optimization accelerates the execution of *read* commands. The number at the top of each bar indicates the speed-up over (vanilla) EPaxos.

With 7 sites, EPaxos executes 1.8K ops/s in the update-heavy workload, whereas Atlas executes 3.2K ops/s when $f = 1$, and 2.8K ops/s when $f = 2$. Although EPaxos and Atlas $f = 2$ have the same fast-quorum size with $n = 7$, the performance gap between the protocols is large for two reasons. First, the key-access distribution in YCSB does not allow EPaxos to take the fast path frequently, since the first 12 records have a 20% chance of getting picked. Due to this, Atlas $f = 2$ takes the fast path for 88% of commands, while EPaxos does so in 70% of cases. This makes the average commit latency of Atlas $f = 2$ lower by 50ms. Second, batches formed in execution with EPaxos are larger than with Atlas $f = 2$ because Atlas prunes unnecessary dependencies (§4): once commands are committed, EPaxos takes on average 147ms to execute them, while Atlas $f = 2$ needs only 30ms. With $f = 1$, Atlas has a longer execution delay of 73ms (this difference between $f = 1$ and $f = 2$ is explained in §5.5). Nevertheless, Atlas $f = 1$ beats Atlas $f = 2$, since it always commits commands after contacting the closest majority, and this compensates for its higher execution delay.

Increasing the percentage of read operations improves the performance of all the protocols because reads do not conflict with other reads. In the read-only workload the performance is simply determined by the quorum size, since all the protocols take the fast path. In this case, both EPaxos and Atlas $f = 2$ execute 3.2K ops/s, while Atlas $f = 1$, which has a smaller fast quorum, executes 4.2K ops/s.

With the NFR optimization and $n = 7$, the protocols execute up to 33% more operations. The highest speedup occurs in the read-only workload, where the protocols execute all commands after a single round trip to the closest majority. In this case, NFR allows EPaxos and Atlas $f = 2$ to match the performance of vanilla Atlas $f = 1$ while maintaining their higher fault-tolerance level. Compared to vanilla EPaxos, Atlas with NFR is up to 1.9x faster with $f = 1$, and 1.6x with $f = 2$. Similar conclusions can be drawn from Figure 9 when the protocols are deployed over 13 sites. Overall, Atlas with NFR outperforms EPaxos by 1.5-2.3x.

6 Related Work

The classical way of implementing SMR is by funneling all commands through a single leader replica [14, 15, 26, 27], which impairs scalability. A way to mitigate this problem is to distribute the leader responsibilities round-robin among replicas, as done in Mencius [21]. However, this makes the system run at the speed of the slowest replica.

Exploiting commutativity to improve the scalability of SMR was first proposed in Generalized Paxos [28] and Generic Broadcast [16]. These protocols still rely on a leader to order concurrent non-commuting commands, which creates a bottleneck.

The closest SMR protocol to ours is EPaxos [24], which is also leaderless and exploits commutativity. We compared Atlas with EPaxos in detail in §3.3. There have been two follow-up protocols to EPaxos, Alvin [33] and Caesar [2]. Atlas compares to these protocols similarly to EPaxos; in particular, both follow-ups have large fast quorums that depend on the overall number of processes only.

Flexible Paxos [12] reduces the size of Paxos Phase 2 quorums to $f + 1$, a technique we also use on the slow path of Atlas. However, this technique is not directly applicable to computing dependencies via fast path, as required by leaderless SMR. To the best of our knowledge, Atlas is the first protocol to reduce the size of fast quorums to $\lfloor \frac{n}{2} \rfloor + f$.

An approach to scaling SMR is to shard the state of the application being replicated and add cross-shard coordination to preserve consistency [3]. Such approaches build on a non-sharded SMR protocol and are hence orthogonal to our proposal: Atlas can be combined with them to scale SMR even further. Protocols such as M2Paxos [29], WPaxos [1] and DPaxos [25] scale up SMR using a variation of the sharding approach. These protocols exploit access locality by optimizing for workloads where commands do not frequently access objects in multiple locations.

There have been recent proposals of SMR protocols that improve scalability using special hardware capabilities, such as low-latency switches or RDMA [8, 19, 35]. However, currently these protocols work within a single data center only.

7 Conclusion

This paper presented Atlas, the first leaderless SMR protocol parameterized with the number of allowed failures. Atlas is designed for planet-scale systems where concurrent site failures are rare. It uses tight quorums, executes a high percentage of the operations within a single round trip and executes quick linearizable reads. As demonstrated empirically with large-scale experiments in Google Cloud Platform, all these innovations pay off in practice: adding new nearby replicas improves client-perceived latency, and expanding to new locations maintains the system performance. Compared to the state of the art, Atlas consistently outperforms existing protocols: it is up to two times faster than Flexible Paxos with identical failure assumptions, and more than doubles the performance of EPaxos in mixed read-write workloads.

Acknowledgments

We thank Lennart Oldenburg for his valuable feedback on early versions of this paper. We also thank our shepherd, Liuba Shrira, and the anonymous reviewers for their comments and suggestions. Vitor Enes was supported by an FCT PhD Fellowship (PD/BD/142927/2018). Tuanir França Rezende and Pierre Sutra were supported by EU H2020 grant No 825184 and ANR grant 16-CE25-0013-04. Alexey Gotsman was supported by an ERC Starting Grant RACCOON. This work was partially supported by the Google Cloud Platform research credits program.

References

[1] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. Multileader WAN Paxos: Ruling the Archipelago with Fast Consensus. *arXiv CoRR*, abs/1703.08905, 2017.

[2] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up Consensus by Chasing Fast Decisions. In *International Conference on Dependable Systems and Networks (DSN)*, 2017.

[3] Carlos Eduardo Benevides Bezerra, Fernando Pedone, and Robbert van Renesse. Scalable State-Machine Replication. In *International Conference on Dependable Systems and Networks (DSN)*, 2014.

[4] Brendan Burns, Brian Grant, David Oppenheimer, Eric A. Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *ACM Queue*, 2016.

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

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Symposium on Cloud Computing (SoCC)*, 2010.

[7] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[8] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at Network Speed. In *Symposium on Software Defined Networking Research (SOSR)*, 2015.

[9] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-Machine Replication for Planet-Scale Systems (Extended Version). *arXiv CoRR*, abs/2003.11789, 2020.

[10] Seth Gilbert and Nancy A. Lynch. Brewer's Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services. *SIGACT News*, 2002.

[11] Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 1990.

[12] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible Paxos: Quorum Intersection Revisited. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2016.

[13] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference (USENIX ATC)*, 2010.

[14] Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *International Conference on Dependable Systems and Networks (DSN)*, 2011.

[15] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 1998.

[16] Leslie Lamport. Generalized Consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[17] Leslie Lamport. Fast Paxos. *Distributed Computing*, 2006.

[18] Leslie Lamport. Lower Bounds for Asynchronous Consensus. *Distributed Computing*, 2006.

[19] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[20] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: Practical Fault Tolerance beyond Crashes. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[21] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building Efficient Replicated State Machine for WANs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[22] Henrique Moniz, João Leitão, Ricardo J. Dias, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. Blotter: Low Latency Transactions for Geo-Replicated Storage. In *International Conference on World Wide Web (WWW)*, 2017.

[23] Iulian Moraru. Egalitarian Distributed Consensus. Technical Report CMU-CS-14-133, Carnegie Mellon University, 2014. PhD Thesis.

[24] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There Is More Consensus in Egalitarian Parliaments. In *Symposium on Operating Systems Principles (SOSP)*, 2013.

[25] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. DPaxos: Managing Data Closer to Users for Low-Latency and Mobile Applications. In *International Conference on Management of Data (SIGMOD)*, 2018.

[26] Brian M. Oki and Barbara Liskov. Viewstamped Replication: A General Primary Copy. In *Symposium on Principles of Distributed Computing (PODC)*, 1988.

[27] Diego Ongaro and John K. Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[28] Fernando Pedone and André Schiper. Generic Broadcast. In *International Symposium on Distributed Computing (DISC)*, 1999.

[29] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Making Fast Consensus Generally Faster. In *International Conference on Dependable Systems and Networks (DSN)*, 2016.

[30] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 1990.

[31] Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Paiva Junqueira. Dynamic Reconfiguration of Primary/Backup Clusters. In *USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[32] Pierre Sutra. On the correctness of Egalitarian Paxos. *Inf. Process. Lett.*, 2020.

[33] Alexandru Turcu, Sebastiano Peluso, Roberto Palmieri, and Binoy Ravindran. Be General and Don't Give Up Consistency in Geo-Replicated Transactional Systems. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2014.

[34] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V. Madhyastha. Near-Optimal Latency Versus Cost Tradeoffs in Geo-Distributed Storage. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[35] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. APUS: Fast and Scalable Paxos on RDMA. In *Symposium on Cloud Computing (SoCC)*, 2017.