Leaderless State-Machine Replication: Specification, Properties, Limits

Tuanir França Rezende
Telecom SudParis, Évry, France
tuanir.franca-rezende@telecom-sudparis.eu

Pierre Sutra
Telecom SudParis, Évry, France
pierre.sutra@telecom-sudparis.eu

Abstract

Modern Internet services commonly replicate critical data across several geographical locations using state-machine replication (SMR). Due to their reliance on a leader replica, classical SMR protocols offer limited scalability and availability in this setting. To solve this problem, recent protocols follow instead a leaderless approach, in which each replica is able to make progress using a quorum of its peers. In this paper, we study this new emerging class of SMR protocols and states some of their limits. We first propose a framework that captures the essence of leaderless state-machine replication (Leaderless SMR). Then, we introduce a set of desirable properties for these protocols: (R)eliability, (O)ptimal (L)atency and (L)oad Balancing. We show that protocols matching all of the ROLL properties are subject to a trade-off between performance and reliability. We also establish a lower bound on the message delay to execute a command in protocols optimal for the ROLL properties. This lower bound explains the persistent chaining effect observed in experimental results.

2012 ACM Subject Classification General and reference → Performance; Software and its engineering → Distributed systems organizing principles; Theory of computation → Distributed computing models

Keywords and phrases Fault Tolerance, State Machine Replication, Consensus

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.24

Related Version A full version of the paper is available at [22], `https://arxiv.org/abs/2008.02512`.

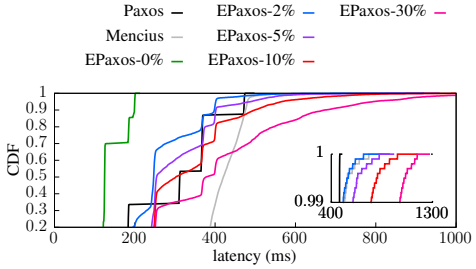
Funding This research is partly funded by the ANR RainbowFS project and the H2020 CloudButton project.

Acknowledgements The authors thank Vitor Enes and Alexey Gotsman for fruitful discussions on Leaderless SMR.

1 Introduction

The standard way of implementing fault-tolerant distributed services is state-machine replication (SMR) [25]. In SMR, a service is defined by a deterministic state machine, and each process maintains its own local copy of the machine. Classical SMR protocols such as Paxos [13] and Raft [20] rely on a leader replica to order state-machine commands. The leader orchestrates a growing sequence of agreements, or consensus, each defining the next command to apply on the state machine. Such a scheme has however clear limitations, especially in a geo-distributed setting. First, it increases latency for clients that are far away from the leader. Second, as the leader becomes a bottleneck or its network gets slower, system performance decreases. Last, this approach harms availability because when the leader fails the whole system cannot serve new requests until an election takes place.

To sidestep the above limitations, a new class of leaderless protocols has recently emerged [18, 19, 3, 8, 26, 7]. These protocols allow any replica to make progress as long as it is able to contact enough of its peers. Mencius [18] pioneered this idea by rotating the ownership of consensus instances. Many other works have followed, and in particular the Egalitarian Paxos



(a) Latency distribution when varying the conflict rate.

	SC	FI	QC	AU	TW
SC		123	25	199	184
FI			120	308	289
QC				202	182
AU					127

(b) Ping distance between sites (in ms).

Figure 1 Performance comparison of EPaxos, Paxos and Mencius – *5 sites: South Carolina (SC), Finland (FI), Canada (QC), Australia (AU), Taiwan (TW, leader); 128 clients per site;* no-op *service.*

(EPaxos) protocol [19]. As Generalized Paxos [14], EPaxos orders only non-commuting, aka. conflicting, state-machine commands. To this end, the protocol maintains at each replica a directed graph that stores the execution constraints between commands. Execution of a command proceeds by linearizing the graph of constraints. In the common case, EPaxos executes a command after two message delays if the fast path was taken, that is, if the replicas spontaneously agree on the constraints, and four message delays otherwise.

Problem statement. Unfortunately the latency of EPaxos may raise in practice well above four message delays. To illustrate this point, we ran an experimental evaluation of EPaxos, Paxos and Mencius in Google Cloud Platform. The results are reported in Figure 1a, where we plot the cumulative distribution function (CDF) of the command latency for each protocol. In this experiment, the system spans five geographical locations distributed around the globe, and each site hosts 128 clients that execute *no-op* commands in closed-loop. Figure 1b indicates the distance between any two sites. The conflict rate among commands varies from 0% to 30%.[1] We measure the latency from the submission of a command to its execution (at steady state).

Two observations can be formulated at the light of the results in Figure 1. First, the tail of the latency distribution in EPaxos is larger than for the two other protocols and it increases with the conflict rate. Second, despite Mencius clearly offering a lower median latency, it does not exhibit such a problem.

Contributions. In this paper, we provide a theoretical framework to understand and explain the above phenomena. We study in-depth this new class of leaderless state-machine replication (Leaderless SMR) protocols and state some of their limits.

Paper Outline. We recall the principles of state-machine replication (§2). Then, we define Leaderless SMR and deconstruct it into basic building blocks (§3). Further, we introduce a set of desirable properties for Leaderless SMR: (R)eliability, (O)ptimal (L)atency and (L)oad Balancing. Protocols that match all of the ROLL properties are subject to a trade-off between performance and reliability. More precisely, in a system of n processes, the ROLL theorem

[1] Each command has a key and any two commands conflict, that is they must be totally ordered by the protocol, when they have the same key. When a conflict rate ρ is applied, each client picks key 42 with probability ρ, and a unique key otherwise.

(§4) states that Leaderless SMR protocols are subject to the inequality $2F + f - 1 \leq n$, where $n - F$ is the size of the fast path quorum and f is the maximal number of tolerated failures. A protocol is ROLL-optimal when F and f cannot be improved according to this inequality. We establish that ROLL-optimal protocols are subject to a chaining effect that affect their performance (§5). As EPaxos is ROLL-optimal and Mencius not, the chaining effect explains the performance results observed in Figure 1. We discuss the implications of this result (§6) then put our work in perspective (§7) before closing (§8).

2 State machine replication

State-machine replication (SMR) allows a set of distributed processes to construct a linearizable shared object. The object is defined by a deterministic state machine together with a set of commands. Each process maintains its own local replica of the machine. An SMR protocol coordinates the execution of commands applied to the state machine, ensuring that the replicas stay in sync. This section recalls the fundamentals of SMR, as well as its generalization that leverages the commutativity of state-machine commands.

2.1 System model

We consider the standard model of wait-free computation in a distributed message-passing system where processes may fail-stop [9]. In [6], the authors extend this framework to include failure detectors. This paper follows such a model of distributed computation. Further details appear in [22].

2.2 Classic SMR

State machine replication is defined over a set of $n \geq 2$ processes Π using a set $\mathcal{C}$ of state-machine commands. Each process p holds a log, that is a totally ordered set of entries that we assume unbounded. Initially, each entry in the log is empty (i.e., $log_p[i] = \perp$ for $i \in \mathbb{N}$), and over time it may include one state-machine command. The operator $(log_p \bullet \mathsf{c})$ *appends* command c to the log, assigning it to the next free entry.

Commands are submitted by the processes that act as proxies on behalf of a set of remote clients (not modeled). A process takes the step $submit(\mathsf{c})$ to submit command c for inclusion in the log. Command c is *decided* once it enters the log at some position i. It is executed against the state machine when all the commands at lower positions ($j < i$) are already executed. When the command is executed, its response value is sent back to the client. For simplicity, we shall consider that two processes may submit the same command.

When the properties below hold during every execution, the above construct ensures that the replicated state machine implements a linearizable shared object.

Validity: A command is decided once and only if it was submitted before.

Stability: If $log_p[i] = \mathsf{c}$ holds at some point in time, it is also true at any later time.

Consistency: For any two processes p and q, if $log_p[i]$ and $log_q[i]$ are both non-empty, then they are equal.

2.3 Generic SMR

In their seminal works, Pedone and Schiper [21] and concurrently Lamport [14] introduce an alternative approach to Classic SMR. They make the key observation that if commands submitted to the state machine commute, then there is no need to order them. Leveraging this, they replace the totally-ordered log used in Classic SMR by a partially-ordered one. We call this approach Generic SMR.

Two commands c and d do not commute when for some state s, applying cd to s differs from applying dc. This means that either both sequences do not lead to the same state, or one of the two commands does not return the same response value in the two sequences. Generic SMR relies on the notion of *conflicts* which captures a safe over-approximation of the non-commutativity of two state-machine commands. In what follows, conflicts are expressed as a binary, non-reflexive and symmetric relation $\asymp$ over $\mathcal{C}$.

In Generic SMR, each variable log_p is a partially ordered log, i.e., a directed acyclic graph [14]. In this graph, vertices are commands and any two conflicting commands have a directed edge between them. We use $G.V$ and $G.E$ to denote respectively the vertices of some partially ordered log G and its edges. The append operator is defined as follows: $G \bullet \mathsf{c} := (G.V \cup \{\mathsf{c}\}, G.E \cup \{(\mathsf{d}, \mathsf{c}) : \mathsf{d} \in G.V \wedge \mathsf{d} \asymp \mathsf{c}\}$. A command is decided once it is in the partially ordered log. As previously, it gets executed once all its predecessors are executed.

For correctness, Generic SMR defines a set of properties over partially ordered logs similar to Classic SMR. Stability is expressed in close terms, using a prefix relation between the logs along time. Consistency requires the existence of a common least upper bound over the partially ordered logs.

To state this precisely, consider two partially ordered logs G and H. G is prefix of H, written $G \sqsubseteq H$, when G is a subgraph of H and for every edge $(\mathsf{a}, \mathsf{b}) \in H.E$, if $\mathsf{b} \in G.V$ then $(\mathsf{a}, \mathsf{b}) \in G.E$. Given a set $\mathcal{G}$ of partially ordered logs, H is an *upper bound* of $\mathcal{G}$ iff $G \sqsubseteq H$ for every G in $\mathcal{G}$. Two logs G and H are *compatible* iff they have a common upper bound.[2] By extension, a set $\mathcal{G}$ of partially ordered logs is compatible iff its elements are pairwise compatible.

Based on the above definitions, we may express Generic SMR using the set of properties below. Validity is identical to Classic SMR and thus omitted.

Stability: For any process p, at any given time log_p is prefix of itself at any later time.

Consistency: The set of all the partially ordered logs is always compatible.

3 Leaderless SMR

Some recent protocols [18, 19] further push the idea of partially ordered log, as proposed in Generic SMR. In a leaderless state-machine replication (Leaderless SMR) protocol, there is no primary process to arbitrate upon the ordering of commands. Instead, any process may decide a command submitted to the replicated service. A command is stable, and thus executable, once the transitive closure of its predecessors is known locally. As this transitive closure can be cyclic, the log is replaced with a directed graph.

This section introduces a high-level framework to better understand Leaderless SMR. In particular, we present the notion of dependency graph and explain how commands are decided. With this framework, we then deconstruct several Leaderless SMR protocols into basic building blocks. Further, three key properties are introduced: Reliability, Optimal Latency and Load Balancing. These properties serve in the follow-up to establish lower bound complexity results for this class of protocols.

[2] In [14], compatibility is defined in terms of least upper bound between two c-structs. For partially ordered logs, the definition provided here is equivalent.

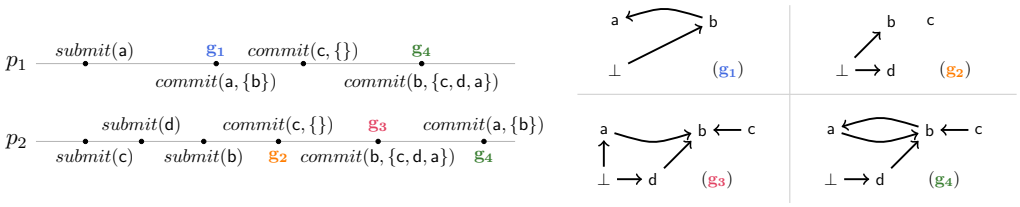


Figure 2 An example run of Leaderless SMR – *(left) processes p_1 and p_2 submit respectively the commands* $\{\mathsf{a}\}$ *and* $\{\mathsf{b}, \mathsf{c}, \mathsf{d}\}$*; (right) the dependencies graphs formed at the two processes.*

3.1 Definition

Leaderless SMR relies on the notion of dependency graph instead of partially ordered log as found in Generic SMR. A *dependency graph* is a directed graph that records the constraints defining how commands are executed. For some command c, the incoming neighbors of c in the dependency graph are its *dependencies*. As detailed shortly, the dependencies are executed either before or together with c.

In Leaderless SMR, a process holds two mapping: *deps* and *phase*. The mapping *deps* is a dependency graph storing a relation from $\mathcal{C}$ to $2^{\mathcal{C}} \cup \{\bot, \top\}$. For a command c, *phase*(c) can take five possible values: pending, abort, commit, stable and execute. All the phases, except execute, correspond to a predicate over *deps*.

Initially, for every command c, *deps*(c) is set to $\bot$. This corresponds to the pending phase. When a process decides a command c, it changes the mapping *deps*(c) to a non-$\bot$ value. Operation *commit*(c, D) assigns D taken in $2^{\mathcal{C}}$ to *deps*(c). Command c gets aborted when *deps*(c) is set to $\top$. In that case, the command is removed from any *deps*(d) and it will not appear later on. Let *deps**(c) be the transitive closure of the *deps* relation starting from $\{\mathsf{c}\}$. Command c is stable once it is committed and no command in *deps**(c) is pending.

Figure 2 depicts an example run of Leaderless SMR that illustrates the above definitions. In this run, process p_1 submits command a, while p_2 submits in order c, d then b. The timeline in Figure 2 indicates the timing of these submissions. It also includes events during which process p_1 and p_2 commits commands. For some of these events, we depict the state of the dependency graph at the process (on the right of Figure 2). As an example, the two processes obtain the graph g_4 at the end of the run. In this graph, a, b and c are all committed, while d is still pending. We have *deps*(a) $= \{\mathsf{b}\}$ and *deps*(b) $= \{\mathsf{a}, \mathsf{d}, \mathsf{c}\}$, with both *deps**($\mathsf{a}$) and *deps**($\mathsf{b}$) equal to $\{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$. Only command c is stable in g_4.

Similarly to Classic and Generic SMR, Leaderless SMR protocols requires that validity holds. In addition, processes must agree on the value of *deps* for stable commands and conflicting commands must see each other. More precisely,

Stability: For each command c, there exists D such that if c is stable then *deps*(c) $= D$.
Consistency: If a and b are both committed and conflicting, then $\mathsf{a} \in$ *deps*(b) or $\mathsf{b} \in$ *deps*(a).

A command c gets executed once it is stable. Algorithm 1 describes how this happens in Leaderless SMR. To execute command c, a process first creates a set of commands, or *batch*, β that execute together with c. This grouping of commands serves to maintain the following invariant:

▶ Invariant 1. *Consider two conflicting commands* c *and* d*. If p executes a batch of commands containing* c *before executing* d*, then $d \notin$ deps$^*(c)$.*

Algorithm 1 Executing command c – code at process p.

```
1: execute(c) :=
2:     pre:  phase(c) = stable
3:     eff:  let β be the largest subset of deps*(c) satisfying ∀d ∈ β. phase(d) = stable
4:           forall d ∈ β ordered by →
5:               phase(d) ← execute
```

Satisfying Invariant 1 implies that if some command d is in batch β, then β also contains its transitive dependencies (line 3 in Algorithm 1). Inside a batch, commands are ordered according to the partial order $\rightarrow$ (line 4). Let $<$ be a canonical total order over $\mathcal{C}$. Then, $\mathsf{c} \rightarrow \mathsf{d}$ holds iff

(i) $\mathsf{c} \in \mathit{deps}^*(\mathsf{d})$ and $\mathsf{d} \notin \mathit{deps}^*(\mathsf{c})$; or

(ii) $\mathsf{c} \in \mathit{deps}^*(\mathsf{d})$, $\mathsf{d} \in \mathit{deps}^*(\mathsf{c})$ and $\mathsf{c} < \mathsf{d}$.

Relation $\rightarrow$ defines the *execution order* at a process. If there is a one-way dependency between two commands, Leaderless SMR plays them in the order of their transitive dependencies; otherwise the algorithm breaks the tie using the arbitrary order $<$. This guarantees the following invariant.

▶ Invariant 2. *Consider two conflicting commands* c *and* d. *If* p *executes* c *before* d *in the same batch, then* $\mathsf{c} \in \mathit{deps}^*(\mathsf{d})$.

Generic and Leaderless SMR are strongly similar. In fact, one may show that Generic SMR reduces to Leaderless SMR without requiring any message exchange. This result is stated in Theorem 1 below, and a proof appears in [22]. Let us observe that such a reduction does not hold between Classic and Generic SMR. Indeed, computing a total order on commuting commands would require processes to communicate.

▶ **Theorem 1.** *Generic SMR reduces to Leaderless SMR.*

However, Theorem 1 offers an incomplete picture of how the two abstractions compare in practice. Indeed, because the dependency graph might be cyclic, Leaderless SMR does not compute an ordering over conflicting commands. Instead, such commands must simply observe one another (Consistency property). This fundamental difference explains the absence of a leader in this class of SMR protocols, a feature that we capture in the next section.

3.2 Deciding commands

In Leaderless SMR, processes have to agree on the dependencies of stable commands. Thus, a subsequent refinement leads to consider a family of consensus objects $(\mathsf{CONS}_\mathsf{c})_{\mathsf{c} \in \mathcal{C}}$ for that purpose. For some command c, processes use CONS_c to decide either the dependencies of c, or the special value $(\top)$ signaling that the command is aborted. This agreement is driven by the command *coordinator* ($\mathit{coord}(\mathsf{c})$), a process initially in charge of submitting the command to the replicated state machine. In a run during which there is no failure and the failure detector behaves perfectly, that is a *nice run*, only $\mathit{coord}(\mathsf{c})$ calls CONS_c.

To create a valid proposal for CONS_c, $\mathit{coord}(\mathsf{c})$ relies on the dependency discovery service (DDS). This shared object offers a single operation $\mathit{announce}(\mathsf{c})$ that returns a pair (D, b), where $D \in 2^\mathcal{C} \cup \{\top\}$ and $b \in \{0, 1\}$ is a flag. When the return value is in $2^\mathcal{C}$, the service suggests to commit the command. Otherwise, the command should be aborted. When the flag is set, the service indicates that a spontaneous agreement occurs. In such a case, the coordinator can directly commit c with the return value of the DDS service and bypass CONS_c; this is called *a fast path*. A *recovery* occurs when command c is announced at a process which is not $\mathit{coord}(\mathsf{c})$.

Algorithm 2 Deciding a command c – code at process p.

```
1: submit(c) :=
2:     pre:  p = coord(c) ∨ coord(c) ∈ 𝒟
3:     eff:  (D, b) ← DDS.announce(c)
4:           if b = false then D ← CONS_c.propose(D)
5:           deps(c) ← D
6:           send(c, deps(c)) to Π \ {p}
7:
8: when recv(c, D)
9:     eff:  deps(c) ← D
```

The DDS service ensures two safety properties. First, if two conflicting commands are announced, they do not miss each other. Second, when a command takes the fast path, processes agree on its committed dependencies.

More formally, assume that $\mathit{announce}_p(\mathsf{c})$ and $\mathit{announce}_q(\mathsf{c}')$ return respectively (D, b) and (D', b') with $D \in 2^{\mathcal{C}}$. Then, the properties of the DDS service are as follows.

Visibility: If $\mathsf{c} \asymp \mathsf{c}'$ and $D' \in 2^{\mathcal{C}}$, then $\mathsf{c} \in D'$ or $\mathsf{c}' \in D$.
Weak Agreement: If $\mathsf{c} = \mathsf{c}'$ and $b = \mathit{true}$, then $D' \in 2^{\mathcal{C}}$ and for every $\mathsf{d} \in D \oplus D'$, every invocation to $\mathit{announce}_r(\mathsf{d})$ returns $(\top, \_)$.

To illustrate these properties, consider that no command was announced so far. In that case $(\varnothing, \mathit{true})$ is a valid response to $\mathit{announce}(\mathsf{c})$. If $\mathit{coord}(\mathsf{c})$ is slow, then a subsequent invocation of $\mathit{announce}(\mathsf{c})$ may either return $\varnothing$, or a non-empty set of dependencies D. However in that case, because the fast path was taken by the coordinator, all the commands in D must eventually abort.

Based on the above decomposition of Leaderless SMR, Algorithm 2 depicts an abstract protocol to decide a command. This algorithm uses a family of consensus objects $((\mathsf{CONS}_\mathsf{c})_{\mathsf{c}\in\mathcal{C}})$, a dependency discovery service (DDS) and a failure detector ($\mathcal{D}$) that returns a set of suspected processes. To submit a command c, a process announces it then retrieves a set of dependencies. This set is proposed to CONS_c if the fast path was not taken (line 4). The result of the slow or the fast path determines the value of the local mapping $\mathit{deps}(\mathsf{c})$ to commit or abort command c. Notice that such a step may also be taken when a process receives a message from one of its peers (line 8).

During a nice run, the system is failure-free and the failure detector service behaves perfectly. As a consequence, only $\mathit{coord}(\mathsf{c})$ may propose a value to CONS_c and this value gets committed. In our view, this feature is the *key characteristic* of Leaderless SMR.

Below, we establish the correctness of Algorithm 2. A proof appears in [22].

▶ **Theorem 2.** *Algorithm 2 implements Leaderless SMR.*

3.3 Examples

To illustrate the framework introduced in the previous sections, we now instantiate well-known Leaderless SMR protocols using it.

Rotating coordinator. For starters, let us consider a rotating coordinator algorithm (e.g., [27]). In this class of protocols, commands are ordered *a priori* by some relation $\ll$. Such an ordering is usually defined by timestamping commands at each coordinator and breaking

ties with the process identities. When *coord*(c) calls DDS.*announce*(c), the service returns a pair (D, false), where D are all the commands prior to c according to $\ll$. Upon recovering a command, the DDS service simply suggests to abort it.

Clock-RSM. This protocol [7] improves on the above schema by introducing a fast path. It also uses physical clocks to speed-up the stabilization of committed commands. Once a command is associated to a timestamp, its coordinator broadcasts this information to the other processes in the system. When it receives such a message, a process waits until its local clock passes the command's timestamp to reply. Once a majority of processes have replied, the DDS service informs the coordinator that the fast path was taken.

Mencius. The above two protocols require a committed command to wait all its predecessors according to $\ll$. Clock-RSM propagates in the background the physical clock of each process. A command gets stable once the clocks of all the processes is higher than its timestamp. Differently, Mencius [18] aborts prior pending commands at the time the command is submitted. In detail, *announce*(c) first approximates D as all the commands prior to c according to $\ll$. Then, command c is broadcast to all the processes in the system. Upon receiving such a message, a process q computes all the commands d smaller than c it is coordinating. If d is not already announced, q stores that d will be aborted. Then, q sends d back to *coord*(c) that removes it from D. The DDS service returns (D, f) with f set to *true* if *coord*(c) received a message from everybody. Upon recovering c, if the command was received the over-approximation based on $\ll$ is returned together with the flag *false*. In case c is unknown, the DDS service suggests to abort it.

EPaxos. In [19], the authors present Egalitarian Paxos (EPaxos), a family of efficient Leaderless SMR protocols. For simplicity, we next consider the variation which does not involve sequence numbers. To announce a command c, the coordinator broadcasts it to a quorum of processes. Each process p computes (and records) the set of commands D_p conflicting with c it has seen so far. A call to *announce*(c) returns $(\cup_p D_p, b)$, with b set to *true* iff processes spontaneously agree on dependencies (i.e., for any p, q, $D_p = D_q$). When a process in the initial quorum is slow or a recovery occurs, c is broadcast to everybody. The caller then awaits for a majority quorum to answer and returns (D, false) such that if at least $\frac{f+1}{2}$ processes answer the same set of conflicts for c, then D is set to this value (with $n = 2f + 1$). Alternatively, if at least one process knows c, the union of the response values is taken. Otherwise, the DDS service suggests to abort c.

Caesar. To avoid cycles in the dependency graph, Caesar [3] orders commands using logical timestamps. Upon submitting a command c, the coordinator timestamps it with its logical clock then it executes a broadcast. As with EPaxos, when it receives c a process p computes the conflicting commands D_p received so far. Then, it awaits until there is no conflicting command d with a higher timestamp than c such that $\mathsf{c} \notin \mathit{deps}(\mathsf{d})$. If such a command exists, p replies to the coordinator that the fast path cannot be taken. The DDS service returns $(\cup_p D_p, b)$, where $b = \mathit{true}$ iff no process disables the fast path.

The above examples show that multiple implementations are possible for Leaderless SMR. In the next section, we introduce several properties of interest to characterize them.

3.4 Core properties

State machine replication helps to mask failures and asynchrony in a distributed system. As a consequence, a first property of interest is the largest number of failures (parameter f) tolerated by a protocol. After f failures, the protocol may not guarantee any progress.[3]

(*Reliability*) In every run, if there are at most f failures, every submitted command gets eventually decided at every correct process.

Leaderless SMR protocols exploit the absence of contention on the replicated service to boost performance. In particular, some protocols are able to execute a command after a single round-trip, which is clearly optimal [16]. To ensure this property, the fast path is taken when there is no concurrent conflicting command. Moreover, the command stabilizes right away, requiring that the DDS service returns only submitted commands.

(*Optimal Latency*) During a nice run, every call to $announce(c)$ returns a tuple (D, b) after two message delays such that

- **(i)** if there is no concurrent conflicting command to c, then b is set to *true*,
- **(ii)** $D \in 2^{\mathcal{C}}$, and
- **(iii)** for every $d \in D$, d was announced before.

The replicas that participate to the fast path vary from one protocol to another. Mencius use all the processes. On the contrary, EPaxos solely contact $\lfloor \frac{3n}{4} \rfloor$ of them (or equivalently, $f + \frac{f+1}{2}$ when $n = 2f + 1$). For some command c, a *fast path quorum* for c is any set of $n - F$ replicas that includes the coordinator of c. Such a set is denoted $FQuorums(c)$ and formally defined as $\{Q \mid Q \subseteq \Pi \wedge coord(c) \in Q \ \wedge |Q| \geq n - F\}$. A protocol has the *Load Balancing* property when it may freely choose fast path quorums to make progress.

(*Load Balancing*) During a nice run, any fast path quorum in $FQuorums(c)$ can be used to announce a command c.

The previous properties are formally defined in [22]. Table 1 indicates how they are implemented by well-known leaderless protocols. The columns 'Reliability' and 'Load Balancing' detail respectively the maximum number of failures tolerated by the protocol and the size of the fast path quorum. Notice that by CAP [11], we have $F, f \leq \lfloor \frac{n-1}{2} \rfloor$ when the protocol matches all of the properties. Table 1 also mentions the optimality of each protocol with respect to the ROLL theorem. This theorem is stated in the next section and establishes a trade-off between fault-tolerance and performance in Leaderless SMR.

4 The ROLL theorem

Reliability, Optimal Latency and Load Balancing are called collectively the ROLL properties. These properties introduce the parameters f and F as key characteristics of a Leaderless SMR protocol. Parameter f translates the reliability of the protocol, stating that progress is guaranteed only if less than f processes crash. Parameter F captures its scalability since, any quorum of $n - F$ processes may be used to order a command. An ideal protocol should strive to minimize $n - F$ while maximizing f.

[3] When f failures occur, the system configuration must change to tolerate subsequent ones. If data is persisted (as in Paxos [13]), the protocol simply stops when more than f failures occurs and awaits that faulty processes are back online.

Table 1 The properties of several leaderless SMR protocols – Min stands for a minority of replicas ($\lfloor \frac{n-1}{2} \rfloor$), Maj a majority ($\lceil \frac{n+1}{2} \rceil$), and LMaj a large majority ($\lfloor \frac{3n}{4} \rfloor$).

Protocols	***Properties***			
	Load Balancing ($n - F$)	Reliability (f)	Optimal Latency	ROLL-optimal
Rotating coord.	0	Min	×	×
Clock-RSM [7]	n	Min	×	×
Mencius [18]	n	Min	√	×
Caesar [3]	$\lceil \frac{3n}{4} \rceil$	Min	√	×
EPaxos [19]	LMaj	Min	√	if $n = 2f + 1$
Alvin [26]	LMaj	Min	√	if $n = 2f + 1$
Atlas [8]	$\lfloor \frac{n}{2} \rfloor + f$	any	√	if $n \in 2\mathbb{N} \cup \{3\} \wedge f = 1$

Unfortunately, we show that there is no free-lunch and that an optimization choice must be made. The ROLL theorem below establishes that $2F + f - 1 \leq n$ must hold. This inequality captures that every protocol must trade scalability for fault-tolerance. EPaxos [19] and Atlas [8] illustrate the two ends of the spectrum of solutions (see Table 1). EPaxos supports that any minority of processes may fail, but requires large quorums. Atlas typically uses small fast path quorums ($\lfloor \frac{n}{2} \rfloor + f$), but exactly handles at most f failures.

Below, we state the ROLL theorem and provide a sketch of proof illustrated in Figure 3. A formal treatment appears in [22].

▶ **Theorem 3** (ROLL). *Consider an SMR protocol that satisfies the ROLL properties. Then, it is true that* $2F + f - 1 \leq n$.

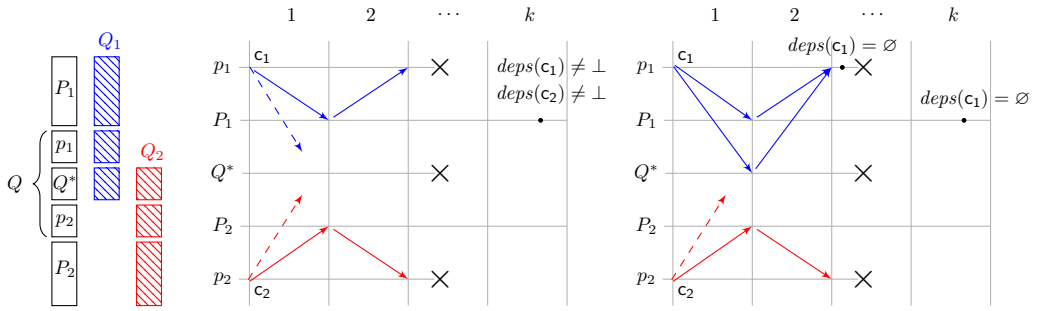
Proof (Sketch). Our proof goes by contradiction, using a round-based reasoning. Let us assume a protocol $\mathcal{P}$ that satisfies all the ROLL properties with $2F + f - 1 > n$. Then, choose two non-commuting commands c_1 and c_2 in $\mathcal{C}$.

As depicted in Figure 3a, the distributed system is partitioned into three sets: P_1 and P_2 are two disjoints sets of $F - 1$ processes, and the remaining $n - 2(F - 1)$ processes form Q. The CAP impossibility result [11] tells us that $2F < n$. As a consequence, there exist at least two distinct processes p_1 and p_2 in Q. We define Q_1 and Q_2 as respectively $P_1 \cup Q \setminus \{p_2\}$ and $P_2 \cup Q \setminus \{p_1\}$. The set Q^* equals $Q \setminus \{p_1, p_2\}$.

Let λ_1 be a nice run that starts from the submission of c_1 by process p_1 during which only Q_1 take steps. Since Q_1 contains $n - F$ processes such a run exists by the Load Balancing property of $\mathcal{P}$. By Optimal Latency, this run lasts two rounds and $deps(\mathsf{c}_1)$ is set to $\varnothing$ at process p_1. Similarly, we may define λ_2 a run in which p_2 announces command c_2 and in which only the processes in Q_2 participate.

Then, consider a run λ_3 in which p_1 and p_2 submit concurrently commands c_1 and c_2. This run is illustrated in Figure 3b. At the end of the first round, the processes in P_1 (respectively, P_2) receive the same messages as in λ_1 (resp., λ_2). At the start of the second round, they reply to respectively p_1 and p_2 as in λ_1 and λ_2. All the other messages sent in the first two rounds are arbitrarily slow. The processes in Q crash at the end of the second round. By Reliability and as $f \geq |Q|$, the commands c_1 and c_2 are stable in λ_3. Let k be the first round at which the two commands are stable at some process $p \in P_1 \cup P_2$.

We now build an admissible run λ_4 of $\mathcal{P}$ as follows. The failure pattern and failure detector history are the same as in λ_3. Commands c_1 and c_2 are submitted concurrently at



(a) Quorums in use. **(b)** Run λ_3. **(c)** Run λ_4.

Figure 3 Illustration of Theorem 3 – *slow messages are omitted.*

the start of λ_4, as in λ_3. In the first two rounds, P_1 receives the same messages as in λ_1 while P_2 receives the same messages as in λ_3. The other messages exchanged during the first two rounds are arbitrarily slow. Figure 3c depicts run λ_4.

Observe that the following claims about λ_4 are true. First, (C1) for p_1, λ_4 is indistinguishable to λ_1 up to round 2. Moreover, (C2) for the processes in $(P_1 \cup P_2)$, λ_4 is indistinguishable to λ_3 up to round k. From (C1), c_1 is stable at p_1 with $deps(\mathsf{c}_1) = \varnothing$. Claim (C2) implies that both c_1 and c_2 are stable at p when round k is reached. By the stability property of Leaderless SMR, process p and p_1 decide the same dependencies for c_1, i.e., $deps(\mathsf{c}_1) = \varnothing$.

A symmetric argument can be made using run λ_2 and a run λ_5, showing that p decides $deps(\mathsf{c}_2) = \varnothing$ in λ_3. It follows that in λ_3, an empty set of dependencies is decided for both commands at process p; a contradiction to the Consistency property. ◀

Theorem 3 captures an inherent trade-off between performance and reliability for ROLL protocols. For instance, tolerating a minority of crashes, requires accessing at least $\lfloor \frac{3n}{4} \rfloor$ processes. This is the setting under which EPaxos operates. On the other hand, if the protocol uses a plain majority quorum in the fast path, it tolerates at most one failure.

4.1 Optimality

A protocol is *ROLL-optimal* when the parameters F and f cannot be improved according to Theorem 3. In other words, they belong to the skyline of solutions [5]. As an example, when the system consists of 5 processes, there is a single such tuple $(F, f) = (2, 2)$. With $n = 7$, there are two tuples in the skyline, $(2, 3)$ and $(3, 2)$. The first one is attained by EPaxos, while Atlas offers the almost optimal solution $(3, 1)$ (see Table 1).

For each protocol, Table 1 lists the conditions under which ROLL-optimality is attained. EPaxos and Alvin are both optimal under the assumption that $n = 2f + 1$. Atlas adjusts the fast path quorums to the value of f, requiring $\lfloor \frac{n}{2} \rfloor + f$ processes to participate. This is optimal when $f = 1$ and either n is even or equals to 3. In the general case, the protocol is within $O(f)$ of the optimal value. As it uses classical Fast Paxos quorums, Caesar is not ROLL-optimal. This is also the case of protocols that contact all of the replicas to make progress, such as Mencius and Clock-RSM. To the best of our knowledge, no protocol is optimal in the general case.

In the next section, we show that ROLL-optimality has a price. More precisely, we establish that by being optimal, a protocol may create an arbitrarily long chain of commands, even during a nice run. This chaining effect may affect adversely the performance of the protocol. We discuss measures of mitigation in §6.

5 Chaining effect

This section shows that a chaining effect may affect ROLL-optimal protocols. It occurs when the chain of transitive dependencies of a command keeps growing after it gets committed. This implies that the committed command takes time to stabilize, thus delaying its execution and increasing the protocol latency.

At first glance, one could think that this situation arises from the asynchrony of the distributed system. As illustrated in Figure 1, this is not the case. We establish that such an effect may occur during "almost" synchronous runs.

The remaining of this section is split as follows. First, we define the notion of chain, that is a dependency-related set of commands. A chain is live when its last command is not stable. To measure how asynchronous a nice run is, we then introduce the principle of k-asynchrony. A run is k-asynchronous when some message is concurrent to both the first and last message of a sequence of k causally-related messages.

At core, our result shows how to inductively add a new link to a live chain during an appropriate 2-asynchronous run of a ROLL-optimal protocol.

5.1 Notion of chain

A chain is a sequence of commands $\mathsf{c}_1 \dots \mathsf{c}_n$ such that for any two consecutive commands $(\mathsf{c}_i, \mathsf{c}_{i+1})$ in the chain, $\mathsf{c}_i \in \mathit{deps}(\mathsf{c}_{i+1})$ at some process. Two consecutive commands (c, d) in a chain form a *link*. For instance, in the dependency graph g_4 (see Figure 2), cba is a chain.

We shall say that a chain is *live* when its first command is not stable yet (at any of the processes). In g_4, this is the case of the chain dba, since command d is still pending ($\mathit{deps}(\mathsf{d}) = \bot$). When a chain is live, the last command in the chain has to wait to ensure a sound execution order across processes. This increases the protocol latency.

5.2 A measure of asynchrony

In a synchronous system [17], processes executes rounds in lock-step. During a round, the messages sent at the beginning are received at the end (provided there is no failure). On the other hand, a partially synchronous system may delay messages for an arbitrary amount of time. In this model, we propose to measure asynchrony by looking at the overlaps between the exchanges of messages. The larger the overlap is, the more asynchronous is the run.

To illustrate this idea, consider the run depicted in Figure 4. During this run, a red message is sent from p_5 to p_4 (bottom left corner of the figure). In the same amount of time p_1 sends a blue message to p_2 which is followed by a green message to p_4. To characterize such an asynchrony, we shall say that the run is 2-asynchronous. This notion is precisely defined below.

▶ **Definition 1** (Path). *A sequence of event $\rho = \mathit{send}_p(m_1)\mathit{recv}_q(m_1)\mathit{send}_q(m_2)\dots\mathit{recv}_t(m_{k\geq 1})$ in a run is called a* path. *We note $\rho[i]$ the i-th message in the path. The number of messages in the path, or its* size, *is denoted $|\rho|$.*

▶ **Definition 2** (Overlapping). *Two messages m and m' are overlapping when their respective events are concurrent.*[4] *By extension, a message m overlaps with a path ρ when it overlaps with both $\rho[1]$ and $\rho[|\rho|]$.*

▶ **Definition 3** (k-asynchrony). *A run λ is k-asynchronous when for every message m, if m overlaps with a path ρ then $|\rho| \leq k$.*

[4] That is, neither $\mathit{recv}(m)$ precedes $\mathit{send}(m')$, nor $\mathit{recv}(m')$ precedes $\mathit{send}(m)$ in real-time.

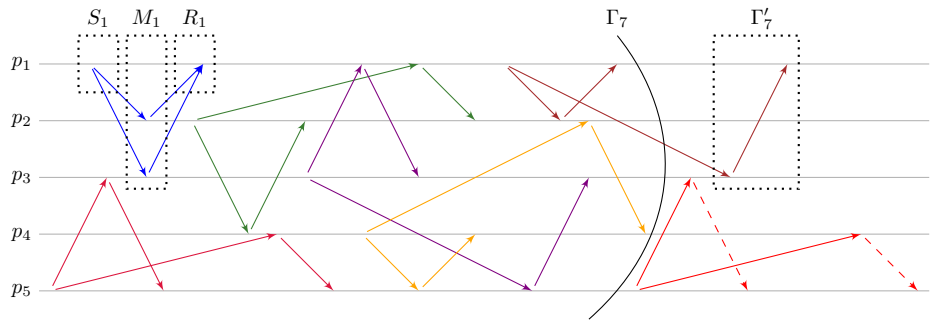


Figure 4 Theorem 4 for $n = 5$ and $k = 7$. The chain $\mathsf{c}_7\mathsf{c}_6\mathsf{c}_5\mathsf{c}_4\mathsf{c}_3\mathsf{c}_2\mathsf{c}_1$ is formed in σ_7. Illustrating the steps S_1, M_1 and R_1 for command c_1, the prefix Γ_7 of σ_7, and the steps Γ'_7.

5.3 Result statement

The theorem below establishes that a ROLL-optimal protocol may create a live chain of arbitrary size during a 2-asynchronous nice run. The full proof appears in [22].

▶ **Theorem 4** (Chaining Effect). *Assume a ROLL-optimal protocol $\mathcal{P}$. For any $k > 0$, there exists a 2-asynchronous nice run of $\mathcal{P}$ containing a live chain of size k.*

Proof (Sketch). The theorem is proved by adding inductively a new link to a live chain of commands created during a nice run. It is illustrated in Figure 4 for a system of five processes when $k = 7$.

The proof is based on the following two key observations about ROLL-optimal protocols. First, during a nice run, the coordinator of a command never rotates. As a consequence, the return value of the DDS service at the coordinator is always the stable value of $deps(c)$. Second, as the protocol satisfies the ROLL properties, a call to $announce(c)$ consists of sending a set of requests to the fast path quorum and receiving a set of replies. As a consequence, its execution can be split into the steps $S_cM_cR_c$, where

(S_c) are the steps taken from announcing c to the sending of the last request at the coordinator;

(R_c) are the steps taken by $coord(c)$ after receiving the first reply until the announcement returns; and

(M_c) are the steps taken during the announcement of c which are neither in S_c, nor in R_c.

By Optimal Latency, this sequence of steps do not create pending messages. As an illustration, the steps S_1, M_1 and R_1 taken to announce command c_1 are depicted in Figure 4.

Leveraging the above two observations, the result is built inductively using a family of k distinct commands $(\mathsf{c}_i)_{i\in[1,k]}$. Each command is associated with a nice run (σ_i), a fast path quorum (Q_i), a subset of $f-1$ processes (P_i), and a process (q_i).

Given a sequence of steps λ and a set of processes Q, let us note $\lambda|Q$ the sub-sequence of steps by Q in λ. We establish that at rank $i > 0$ the following property $\mathfrak{P}(i)$ holds: There exists a 2-asynchronous run σ_i of the form $\Gamma_i S_i(M_i|P_i)\Gamma'_i(M_i|Q_i \setminus P_i)R_i$ such that

(1) $proc(\Gamma'_i) \cap Q_i = P_i$;

(2) every path in Γ'_i is as most of size one;

(3) no message is pending in σ_i; and

(4) σ_i contains a chain $\mathsf{c}_i\mathsf{c}_{i-1}\cdots\mathsf{c}_1$.

Figure 4 depicts the run σ_7, its prefix Γ_7 and the steps Γ'_7.

Starting from $\mathfrak{P}(i)$, we establish $\mathfrak{P}(i+1)$ as follows. First we show that σ_{i+1} as $\Gamma_{i+1}S_{i+1}(M_{i+1}|P_{i+1})\Gamma'_{i+1}(M_{i+1}|Q_{i+1} \setminus P_{i+1})R_{i+1}$, where $\Gamma_{i+1} = \Gamma_i S_i(M_i|P_i)\Gamma'_i$, and $\Gamma'_{i+1} = (M_i|Q_i \setminus P_i)R_i$ is a nice run.

At rank $i+1$, item (1) is proved with appropriate definitions of the quorums (Q_i and Q_{i+1}), and the sub-quorum (P_i). For instance, in Figure 4, the command c_1 and c_2 have respectively $\{p_1, p_2, p_3\}$ and $\{p_3, p_4, p_5\}$ for fast path quorums. The sub-quorum P_2 is set to the intersection of Q_1 and Q_2, that is $\{p_3\}$. Item (2) follows from the definition of Γ'_{i+1}. The Load-Balancing property implies that (3) holds. A case analysis can then show that σ_{i+1} is 2-asynchronous. It relies on the fact that the $(SMR)_{i+1}$ steps create no pending message and the induction property $\mathfrak{P}(i)$.

To prove that a new link was added, we show that σ_{i+1} is indistinguishable to $coord(\mathsf{c_i})$ to a run in which $\mathsf{c_{i+1}}$ gets committed while missing $\mathsf{c_i}$. Going back to Figure 4, observe that the coordinator of $\mathsf{c_6}$ does not know that the replies of p_2 for command $\mathsf{c_5}$ causally precedes the replies of p_4 to $\mathsf{c_7}$. As a consequence, it must add $\mathsf{c_7}$ to the return value of $DDS.announce(\mathsf{c_6})$.

Finally, to obtain a live chain of size k, it suffices to consider the prefix of σ_{i+1} which does not contain the replies of the fast path quorum. In Figure 4, this corresponds to omitting the dashed messages that contain the reply to the announcement of $\mathsf{c_7}$ ◀

6 Discussion

Leaderless SMR offers appealing properties with respect to leader-driven approaches. Protocols are faster in the best case, suffer from no downtime when the leader fails, and distribute the load among participants. For instance, Figure 1 shows that EPaxos is strictly better than Paxos when there is no conflict. However, the latency of a command is strongly related to its dependencies in this family of SMR protocols. Going back to Figure 1, the bivariate correlation between the latency of a command and the size of the batch with which it executes is greater than 0.7.

Several approaches are possible to mitigate the chaining effect established in Theorem 4. Moraru et al. [19] propose that pure writes (i.e., commands having no response value) return once they are committed.[5] In [8], the authors observe that as each read executes at a single process, they can be excluded from the computation of dependencies. A third possibility is to wait until prior commands return before announcing a new one. However, in this case, it is possible to extend Theorem 4 by rotating the command coordinators to establish that a chain of size n can form.

In ROLL, the Load-Balancing and Optimal-Latency properties constrain the form of the DDS service. More precisely, in a contention-free case, executing the service must consist in a back-and-forth between the command coordinator and the fast path quorum. A weaker definition would allow some messages to be pending when *announce* returns. In this case, it is possible to sidestep the ROLL theorem provided that the system is synchronous: When replying to an announcement a process first sends its reply to the other fast path quorum nodes. The fast path is taken by merging all of the replies. Since the system is synchronous, a process recovering a command will retrieve all the replies at any node in the fast path quorum. Note that under this weaker definition, the ROLL theorem (Theorem 3) still applies in a partially synchronous model. Moreover, a chaining effect (Theorem 4) is also possible, but it requires more asynchrony during a nice run.

[5] In fact, it is possible to return even earlier, at the durable signal, that is once $f+1$ processes have received the command. To ensure linearizability, a later read must however wait for all the prior (conflicting or not) preceding writes.

7 Related work

Protocols. Early leaderless solutions to SMR include rotating coordinators and deterministic merge, aka. collision-fast, protocols. We cover the first class of protocols in §3.3. In a collision-fast protocol [1, 24], processes replicate an infinite array of vector consensus instances. Each vector consensus corresponds to a round. During a round, each process proposes a command (or a batch) to its consensus instance in the vector. If the process is in late, its peers may take over the instance and propose an empty batch of commands. Commands are executed according to their round numbers, applying an arbitrary ordering per round. The size of the vector can change dynamically, adapting to network conditions and/or the application workload. This technique is also used in Paxos Commit [12].

When the ordering is fixed beforehand, processes must advance at the same pace. To fix this issue, Mencius [18] includes a piggy-back mechanism that allows a process to bail out its instances (i.e., proposing implicitly an empty batch). Clock-RSM [7] follows a similar schema, using physical clocks to bypass explicit synchronization in the good cases.

With the above protocols, commands still get delayed by slow processes. Avoiding this so-called delayed commit problem [18] requires to dynamically discover dependencies at runtime. This is the approach introduced in Zieliński's optimistic generic broadcast [29] and EPaxos [19]. Here, as well as in [8], replicas agree on a fully-fledged dependency graph. Caesar [3] uses timestamps to avoid cycles in the graph. However, even in contention-free cases, committing a command can take two round trips. In our classification (see Table 1), this protocol does not have Optimal Latency.

Deconstruction. In [4], the authors introduce the dependency-set and map-agreement algorithms. The two services allow respectively to gather dependencies and agree upon them. A similar decomposition is proposed in [28]. Compared to these prior works, our framework includes the notion of fast path and distinguishes committed and stable commands. An agreement between the processes is necessary only eventually and on the stable part of the dependency graph. This difference allows to capture a wider spectrum of protocols. Our dependency discovery service (DDS) is reminiscent of an adopt-commit object [10] that allows processes to reach a weak agreement. In our case, when the fast path flag is set, processes may disagree on at most the aborted dependencies of a command.

Complexity. Multiple works study the complexity of consensus, the key underlying building block of SMR. Lamport [16] proves several lower bounds on the time and space complexity of this abstraction. The Hyperfast Learning theorem establishes that consensus requires one round-trip in the general case. This explains why we call optimal protocols that return after two message delays. The Fast Learning theorem requires that $n > 2F + f$. This result explains the trade-off between fault-tolerance and performance in Fast Paxos [15]. However, it does not readily apply to Leaderless SMR because only coordinator-centric quorums are fast in that case. For instance, EPaxos is able to run with $F = 1$ and $f = 1$ in a 3-process system. The ROLL theorem (§4) accurately captures this difference.

Traditional complexity measures for SMR and consensus (e.g., the latency degree [23]) consider contention-free and/or perfectly synchronous scenarios. In [2], the authors study the complexity of SMR over long runs. The paper shows that completing an SMR command can be more expensive than solving a consensus instance. Their complexity measure is different from ours and given in terms of synchronous rounds. In §5, we show that in an almost synchronous scenario, contention may create arbitrarily long chains in Leaderless SMR. We discuss mitigation measures in §6.

8 Conclusion

This paper introduces a framework to decompose leaderless state-machine replication (Leaderless SMR) protocols. The framework allows to break down representative protocols into two simple building blocks: a dependency discovery service and a consensus service. We then define a set of desirable properties for Leaderless SMR: (R)eliability, (O)ptimal (L)atency and (L)oad balancing. Protocols matching all of these properties satisfy the inequality $2F + f - 1 \leq n$, where n is number of processes, f the maximum number of failures tolerated by the protocol, and $n - F$ the size of the fast path quorum. Further, we establish that protocols that optimally solve this inequality suffer from a chaining effect. This effect explains the tail latency of some Leaderless SMR protocols in real-world deployments.

References

1 Marcos Kawazoe Aguilera and Robert E. Strom. Efficient atomic broadcast using deterministic merge. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, page 209–218, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/343477.343620.

2 Karolos Antoniadis, Rachid Guerraoui, Dahlia Malkhi, and Dragos-Adrian Seredinschi. State machine replication is more expensive than consensus. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 7:1–7:18, 2018. doi:10.4230/LIPIcs.DISC.2018.7.

3 Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up consensus by chasing fast decisions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 49–60, 2017.

4 Marijke H. L. Bodlaender, Magnús M. Halldórsson, Christian Konrad, and Fabian Kuhn. Brief announcement: Local independent set approximation. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 93–95, 2016. doi:10.1145/2933057.2933068.

5 Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*, page 421–430, USA, 2001. IEEE Computer Society.

6 T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Communications of the ACM*, 43(2):225–267, 1996. URL: http://www.acm.org/pubs/toc/Abstracts/jacm/226647.html.

7 Jiaqing Du, Daniele Sciascia, Sameh Elnikety, Willy Zwaenepoel, and Fernando Pedone. Clock-RSM: Low-Latency Inter-datacenter State Machine Replication Using Loosely Synchronized Physical Clocks. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 343–354, 2014. doi:10.1109/DSN.2014.42.

8 Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3342195.3387543.

9 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. doi:10.1145/3149.214121.

10 Eli Gafni. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, PODC '98, pages 143–152, New York, NY, USA, 1998. ACM.

11 Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. doi:10.1145/564585.564601.

12 Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, March 2006. doi:10.1145/1132863.1132867.

13 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

14 Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

15 Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, October 2006.

16 Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, 2006.

17 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

18 Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for wans. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 369–384, 2008.

19 Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 358–372, 2013.

20 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 305–320, 2014.

21 Fernando Pedone and André Schiper. Generic broadcast. In *International Symposium on Distributed Computing (DISC)*, pages 94–108, 1999.

22 Tuanir França Rezende and Pierre Sutra. Leaderless state-machine replication: Specification, properties, limits (extended version), 2020. arXiv:2008.02512.

23 André Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distrib. Comput.*, 10(3):149–157, April 1997. doi:10.1007/s004460050032.

24 R. Schmidt, L. Camargos, and F. Pedone. Collision-fast atomic broadcast. In *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, pages 1065–1072, 2014.

25 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

26 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)*, pages 33–48, 2014.

27 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, SRDS '09, page 135–144, USA, 2009. IEEE Computer Society. doi:10.1109/SRDS.2009.36.

28 Michael Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M. Hellerstein, and Ion Stoica. "bipartisan paxos: A family of fast, leaderless, modular state machine replication protocols". preprint on webpage at https://mwhittaker.github.io/publications/bipartisan_paxos.pdf.

29 Piotr Zieliński. Optimistic generic broadcast. In Pierre Fraigniaud, editor, *Distributed Computing*, pages 369–383, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.