Efficient Replication via Timestamp Stability

Vitor Enes  
INESC TEC and University of Minho

Carlos Baquero  
INESC TEC and University of Minho

Alexey Gotsman  
IMDEA Software Institute

Pierre Sutra  
Télécom SudParis

Abstract
Modern web applications replicate their data across the globe and require strong consistency guarantees for their most critical data. These guarantees are usually provided via state-machine replication (SMR). Recent advances in SMR have focused on leaderless protocols, which improve the availability and performance of traditional Paxos-based solutions. We propose Tempo – a leaderless SMR protocol that, in comparison to prior solutions, achieves superior throughput and offers predictable performance even in contended workloads. To achieve these benefits, Tempo timestamps each application command and executes it only after the timestamp becomes stable, i.e., all commands with a lower timestamp are known. Both the timestamping and stability detection mechanisms are fully decentralized, thus obviating the need for a leader replica. Our protocol furthermore generalizes to partial replication settings, enabling scalability in highly parallel workloads. We evaluate the protocol in both real and simulated geo-distributed environments and demonstrate that it outperforms state-of-the-art alternatives.

CCS Concepts: • Theory of computation → Distributed algorithms.

Keywords: Fault tolerance, Consensus, Geo-replication.

ACM Reference Format:

1 Introduction
Modern web applications are routinely accessed by clients all over the world. To support such applications, storage systems need to replicate data at different geographical locations while providing strong consistency guarantees for the most critical data. State-machine replication (SMR) [39] is an approach for providing such guarantees used by a number of systems [8, 15, 22, 27, 40, 45]. In SMR, a desired 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 to ensure that the system is linearizable [19], i.e., behaves as if commands are executed sequentially by a single site.

Traditional SMR protocols, such as Paxos [29] and Raft [35], rely on a distinguished leader site that defines the order in which client commands are executed at the replicas. Unfortunately, this site is a single point of failure and contention, and a source of higher latency for clients located far from it. Recent efforts to improve SMR have thus focused on leaderless protocols, which distribute the task of ordering commands among replicas and thus allow a client to contact the closest replica instead of the leader [1, 5, 14, 31, 32, 43]. Compared to centralized solutions, leaderless SMR offers lower average latency, fairer latency distribution with respect to client locations, and higher availability.

Leaderless SMR protocols also generalize to the setting of partial replication, where the service state is split into a set of partitions, each stored at a group of replicas. A client command can access multiple partitions, and the SMR protocol ensures that the system is still linearizable, i.e., behaves as if the commands are executed by a single machine storing a complete service state. This approach allows implementing services that are too big to fit onto a single machine. It also enables scalability, since commands accessing disjoint sets of partitions can be executed in parallel. This has been demonstrated by Janus [33] which adapted a leaderless SMR protocol called Egalitarian Paxos (EPaxos) [32] to the setting of partial replication. The resulting protocol provided better performance than classical solutions such as two-phase commit layered over Paxos.

Unfortunately, all existing leaderless SMR protocols suffer from drawbacks in the way they order commands. Some protocols [1, 5, 14, 32] maintain explicit dependencies between commands: a replica may execute a command only after all its dependencies get executed. These dependencies may form arbitrary long chains. As a consequence, in theory the protocols do not guarantee progress even under a synchronous network. In practice, their performance is unpredictable, and in particular, exhibits a high tail latency [5, 37]. Other protocols [11, 31] need to contact every replica on the critical path of each command. While these protocols guarantee progress...
under synchrony, they make the system run at the speed of the slowest replica.

All of these drawbacks carry over to the setting of partial replication where they are aggravated by the fact that commands span multiple machines.

In this paper we propose Tempo, a new leaderless SMR protocol that lifts the above limitations while handling both full and partial replication settings. Tempo guarantees progress under a synchronous network without the need to contact all replicas. It also exhibits low tail latency even in contended workloads, thus ensuring predictable performance. Finally, it delivers superior throughput than prior solutions, such as EPaxos and Janus. The protocol achieves all these benefits by assigning a scalar timestamp to each command and executing commands in the order of these timestamps. To determine when a command can be executed, each replica waits until the command’s timestamp is stable, i.e., all commands with a lower timestamp are known. Ordering commands in this way is used in many protocols [1, 9, 11, 28]. A key novelty of Tempo is that both timestamping and stability detection are fault-tolerant and fully decentralized, which preserves the key benefits of leaderless SMR.

In more detail, each Tempo process maintains a local clock from which timestamps are generated. In the case of full replication, to submit a command a client sends it to the closest process, which acts as its coordinator. The coordinator computes a timestamp for the command by forwarding it to a quorum of replicas, each of which makes a timestamp proposal, and taking the maximum of these proposals. If enough replicas in the quorum make the same proposal, then the timestamp is decided immediately (fast path). If not, the coordinator does an additional round trip to the replicas to persist the timestamp (slow path); this may happen when commands are submitted concurrently. Thus, under favorable conditions, the replica nearest to the client decides the command’s timestamp in a single round trip.

To execute a command, a replica then needs to determine when its timestamp is stable, i.e., it knows about all commands with lower timestamps. The replica does this by gathering information about which timestamp ranges have been used up by each replica, so that no more commands will get proposals in these ranges. This information is piggy-backed on replicas’ messages, which often allows a timestamp of a command to become stable immediately after it is decided.

The above protocol easily extends to partial replication: in this case a command’s timestamp is the maximum over the timestamps computed for each of the partitions it accesses.

We evaluate Tempo in three environments: a simulator, a controlled cluster environment and using multiple regions in Amazon EC2. We show that Tempo improves throughput over existing SMR protocols by 1.8-5.1x, while lowering tail latency with respect to prior leaderless protocols by an order of magnitude. This advantage is maintained in partial replication, where Tempo outperforms Janus by 1.2-16x.

2 Partial State-Machine Replication

We consider a geo-distributed system where processes may fail by crashing, but do not behave maliciously. State-machine replication (SMR) is a common way of implementing fault-tolerant services in such a system [39]. In SMR, the service is defined as a deterministic state machine accepting commands $C$. Each process maintains a replica of the machine and receives commands from clients, external to the system. An SMR protocol coordinates the execution of commands at the processes, ensuring that they stay in sync.

We consider a general version of SMR where each process replicates only a part of the service state – partial SMR (PSMR) [20, 33, 38]. We assume that the service state is divided into partitions, so that each variable defining the state belongs to a unique partition. Partitions are arbitrarily fine-grained: e.g., just a single state variable. Each command accesses one or more partitions. We assume that a process replicates a single partition, but multiple processes may be co-located at the same machine. Each partition is replicated at $r$ processes, of which at most $f$ may fail. Following Flexible Paxos [21], $f$ can be any value such that $1 \leq f \leq \lfloor \frac{n-1}{r} \rfloor$. This allows using small values of $f$ regardless of the replication factor $r$, which is appropriate in geo-replication [8, 14]. We write $\Pi_p$ for the set of all the processes replicating a partition $p$, $\Pi_c$ for the set of processes that replicate the partitions accessed by a command $c$, and $\Pi$ for the set of all processes.

A PSMR protocol allows a process $i$ to submit a command $c$ on behalf of a client. For simplicity, we assume that each command is unique and the process submitting it replicates one of the partitions it accesses: $i \in \Pi_c$. For each partition $p$ accessed by $c$, the protocol then triggers an upcall $execute_p(c)$ at each process storing $p$, asking it to apply $c$ to the local state of partition $p$. After $c$ is executed by at least one process in each partition it accesses, the process that submitted the command aggregates the return values of $c$ from each partition and returns them to the client.

PSMR ensures the highest standard of consistency of replicated data – linearizability [19] – which provides an illusion that commands are executed sequentially by a single machine storing a complete service state. To this end, a PSMR protocol has to satisfy the following specification. Given two commands $c$ and $d$, we write $c \rightarrow_i d$ when they access a common partition and $c$ is executed before $d$ at some process $i \in \Pi_c \cap \Pi_d$. We also define the following real-time order: $c \rightarrow d$ when the command $c$ returns before the command $d$ was submitted. Let $\rightarrow = (\bigcup_{i=1}^{n} \rightarrow_i) \cup \sim$. A PSMR protocol ensures the following properties:

Validity. If a process executes some command $c$, then it executes $c$ at most once and only if $c$ was submitted before.

Ordering. The relation $\rightarrow$ is acyclic.

Liveness. If a command $c$ is submitted by a non-faulty process or executed at some process, then it is executed at all non-faulty processes in $\Pi_c$. 

The Ordering property ensures that commands are executed in a consistent manner throughout the system [18]. For example, it implies that two commands, both accessing the same two partitions, cannot be executed at these partitions in contradictory orders. As usual, to ensure Liveness we assume that the network is eventually synchronous, and in particular, that message delays between non-failed processes are eventually bounded [12].

PSMR is expressive enough to implement a wide spectrum of distributed applications. In particular, it directly allows implementing one-shot transactions, which consist of independent pieces of code (such as stored procedures), each accessing a different partition [23, 30, 33]. It can also be used to construct general-purpose transactions [33, 41].

3 Single-Partition Protocol

For simplicity, we first present the protocol in the case when there is only a single partition, and cover the general case in §4. We start with an overview of the single-partition protocol.

To ensure the Ordering property of PSMR, TempO assigns a scalar timestamp to each command. Processes execute commands in the order of these timestamps, thus ensuring that processes execute commands in the same order. To submit a command, a client sends it to a nearby process which acts as the coordinator for the command. The coordinator is in charge of assigning a timestamp to the command and communicating this timestamp to all processes. When a process finds out about the command’s timestamp, we say that the process commits the command. If the coordinator is suspected to have failed, another process takes over its role through a recovery mechanism (§5). TempO ensures that, even in case of failures, processes agree on the timestamp assigned to the command, as stated by the following property.

Property 1 (Timestamp agreement). Two processes cannot commit the same command with different timestamps.

A coordinator computes a timestamp for a command as follows (§3.1). It first forwards the command to a fast quorum of \( \lfloor \frac{n}{2} \rfloor + f \) processes, including the coordinator itself. Each process maintains a Clock variable. When the process receives a command from the coordinator, it increments Clock and replies to the coordinator with the new Clock value as a timestamp proposal. The coordinator then takes the highest proposal as the command’s timestamp. If enough processes have made such a proposal, the coordinator considers the timestamp decided and takes the fast path: it just communicates the timestamp to the processes, which commit the command. The protocol ensures that the timestamp can be recovered even if the coordinator fails, thus maintaining Property 1. Otherwise, the coordinator takes the slow path, where it stores the timestamp at a slow quorum of \( f + 1 \) processes using a variant of Flexible Paxos [21]. This ensures that the timestamp survives any allowed number of failures. The slow path may have to be taken in cases when commands are submitted concurrently to the same partition (however, recall that partitions may be arbitrarily fine-grained).

Since processes execute committed commands in the timestamp order, before executing a command a process must know all the commands that precede it.

Property 2 (Timestamp stability). Consider a command \( c \) committed at \( i \) with timestamp \( t \). Process \( i \) can only execute \( c \) after its timestamp is stable, i.e., every command with a timestamp lower or equal to \( t \) is also committed at \( i \).

To check the stability of a timestamp \( t \) (§3.2), each process \( i \) tracks timestamp proposals issued by other processes. Once the Clocks at any majority of the processes pass \( t \), process \( i \) can be sure that new commands will get higher timestamps: these are computed as the maximal proposal from at least a majority, and any two majorities intersect. Process \( i \) can then use the information gathered about the timestamp proposals from other processes to find out about all the commands that have got a timestamp lower than \( t \).

3.1 Commit Protocol

Algorithm 1 specifies the single-partition commit protocol at a process \( i \) replicating a partition \( p \). We assume that self-addressed messages are delivered immediately. A command \( c \in C \) is submitted by a client by calling submit\((c)\) at a process \( i \) that replicates a partition accessed by the command (line 1). Process \( i \) then creates a unique identifier \( id \in D \) and a mapping \( Q \) from a partition accessed by the command to the fast quorum to be used at that partition. Because we consider a single partition for now, in what follows \( Q \) contains only one fast quorum, \( Q[p] \). Finally, process \( i \) sends MSubmit\((id, c, Q)\) to a set of processes \( I_i \), which in the single-partition case simply denotes \{\( i \)\}.

A command goes through several phases at each process: from the initial phase \textscstart, to a \textsccommit phase once the command is committed, and an \textscexecute phase once it is executed. We summarize these phases and allowed phase transitions in Figure 1. A mapping phase at a process tracks the progress of a command with a given identifier through phases. For brevity, the name of the phase written in lower case denotes all the commands in that phase, e.g., \textsc{start} = \{ \( id \in D \mid \text{phase}[id] = \text{start} \). We also define \textscpending as follows: \textscpending = \textscpayload \cup \textscpropose \cup \textscrecover \cup \textscrecovery.

Start phase. When a process receives an MSubmit message, it starts serving as the command coordinator (line 5). The coordinator first computes its timestamp proposal for the command as \( \text{Clock} + 1 \). After computing the proposal, the coordinator sends an MPPropose message to the fast quorum \( Q[p] \) and an MPayload message to the remaining processes. Since the fast quorum contains the coordinator, the coordinator also sends the MPPropose message to itself. As mentioned earlier, self-addressed messages are delivered immediately.
Algorithm 1: Commit protocol at process $i \in I_p$.

1. submit($c$)
2. **pre:** $i \in I_c$
3. id $\leftarrow$ next_id(); $Q \leftarrow$ fast_quorums($i, I_c$)
4. send MSubmit(id, c, Q) to $I^u_c$
5. receive MSubmit(id, c, Q)
6. $t \leftarrow$ Clock + 1
7. send MPpropose(id, c, Q, t) to $Q[p]$
8. send MPayload(id, c, Q) to $I_p \setminus Q[p]$
9. receive MPayload(id, c, Q)
10. **pre:** id $\in$ start
11. cmd[id] $\leftarrow$ c; quorums[id] $\leftarrow Q$; phase[id] $\leftarrow$ PAYLOAD
12. receive MPpropose(id, c, Q, t) from j
13. **pre:** id $\in$ start
14. cmd[id] $\leftarrow$ c; quorums[id] $\leftarrow Q$; phase[id] $\leftarrow$ PROPOSE
15. ts[id] $\leftarrow$ proposal(id, t)
16. send MPproposeAck(id, ts[id]) to j
17. receive MPproposeAck(id, t, j) from $Q \setminus \{j\}$
18. **pre:** id $\in$ propose $\land Q = $ quorums[id][p]
19. $t \leftarrow$ max($t_j \mid j \in Q$)
20. if count(t) $\geq f$ then send MCommit(id, t) to $I_{cmd}[id]$
21. else send MConsensus(id, t, i) to $I_p$
22. receive MCommit(id, t)
23. **pre:** id $\in$ pending
24. ts[id] $\leftarrow$ t; phase[id] $\leftarrow$ COMMIT
25. bump(ts[id])
26. receive MConsensus(id, t, b) from j
27. **pre:** bal[id] $\leq b$
28. ts[id] $\leftarrow$ t; bal[id] $\leftarrow b$; abal[id] $\leftarrow b$
29. bump(t)
30. send MConsensusAck(id, b) to j
31. receive MConsensusAck(id, b) from Q
32. **pre:** bal[id] $= b \land |Q| = f + 1$
33. send MCommit(id, ts[id]) to $I_{cmd}[id]$
34. proposal(id, m)
35. $t \leftarrow$ max(m, Clock + 1)
36. Attached $\leftarrow$ Attached $\cup \{(i, u) \mid \text{Clock} + 1 \leq u \leq t - 1\}$
37. Attached[id] $\leftarrow$ $\{(i, t)\}$
38. Clock $\leftarrow t$
39. **return** t
40. bump(t)
41. $t \leftarrow$ max(t, Clock)
42. Attached $\leftarrow$ Attached $\cup \{(i, u) \mid \text{Clock} + 1 \leq u \leq t\}$
43. Clock $\leftarrow t$

Payload phase. Upon receiving an MPayload message (line 9), a process simply saves the command payload in a mapping cmd and sets the command’s phase to PAYLOAD. It also saves $Q$ in a mapping quorums. This is necessary for the recovery mechanism to know the fast quorum used for the command (§5).

Figure 1. Command journey through phases in TEMPO.

**Propose phase.** Upon receiving an MPpropose message (line 12), a fast-quorum process also saves the command payload and fast quorums, but sets its phase to PROPOSE. Then the process computes its own timestamp proposal using the function proposal and stores it in a mapping ts. Finally, the process replies to the coordinator with an MPproposeAck message, carrying the computed timestamp proposal.

The function proposal takes as input an identifier id and a timestamp $m$ and computes a timestamp proposal as $t = \max(m, \text{Clock} + 1)$, so that $t \geq m$ (line 35). The function bumps the Clock to the computed timestamp $t$ and returns $t$ (lines 38-39); we explain lines 36-37 later. As we have already noted, the coordinator computes the command’s timestamp as the highest of the proposals from fast-quorum processes. Proactively taking the max between the coordinator’s proposal $m$ and Clock + 1 in proposal ensures that a process’s proposal is at least as high as the coordinator’s; as we explain shortly, this helps recovering timestamps in case of coordinator failure.

**Commit phase.** Once the coordinator receives an MPproposeAck message from all the processes in the fast quorum $Q = Q[p]$ (line 17), it computes the command’s timestamp as the highest of all timestamp proposals: $t = \max(t_j \mid j \in Q)$. Then the coordinator decides to either take the fast path (line 20) or the slow path (line 21). Both paths end with the coordinator sending an MCommit message containing the command’s timestamp. Since $|Q| = \lceil \frac{f}{p} \rceil + f$ and $f \geq 1$, we have the following property which ensures that a committed timestamp is computed over (at least) a majority of processes.

**Property 3.** For any message MCommit(id, t), there is a set of processes $Q$ such that $|Q| \geq \lceil \frac{f}{p} \rceil + 1$ and $t = \max(t_j \mid j \in Q)$, where $t_j$ is the output of function proposal(id, _) previously called at process $j \in Q$.

This property is also preserved if $t$ is computed by a process performing recovery in case of coordinator failure (§5).

Once a process receives an MCommit message (line 22), it saves the command’s timestamp in ts[id] and moves the command to the COMMIT phase. It then bumps the Clock to the committed timestamp using a function bump (line 40). We next explain the fast and slow paths, as well as the conditions under which they are taken.

**Fast path.** The fast path can be taken if the highest proposal $t$ is made by at least $f$ processes. This condition is expressed
by count(t) ≥ f in line 20, where count(t) = |{j ∈ Q | t_j = t}|. If the condition holds, the coordinator immediately sends an MCommit message with the computed timestamp. The protocol ensures that, if the coordinator fails before sending all the MCommit messages, t can be recovered as follows. First, the condition count(t) ≥ f ensures that the timestamp t can be obtained without f − 1 fast-quorum processes (e.g., if they fail) by selecting the highest proposal made by the remaining quorum members. Moreover, the proposal by the coordinator is also not necessary to obtain t. This is because fast-quorum processes only propose timestamps no lower than the coordinator’s proposal (line 15). As a consequence, the coordinator’s proposal is only the highest proposal t when all processes propose the same timestamp, in which case a single process suffices to recover t. It follows that t can be obtained without f fast-quorum processes including the initial coordinator by selecting the highest proposal sent by the remaining (⌈f/2⌉ + f) − f = ⌈f/2⌉ quorum members. This observation is captured by the following property.

Property 4. Any timestamp committed on the fast path can be obtained by selecting the highest proposal sent in MPropose by at least ⌈f/2⌉ fast-quorum processes distinct from the initial coordinator.

Fast path examples. Table 1 contains several examples that illustrate the fast-path condition of Tempo and Property 4. All examples consider r = 5 processes. We highlight timestamp proposals in bold. Process A acts as the coordinator and sends 6 in its MPropose message. The fast quorum Q is {A, B, C} when f = 1 and {A, B, C, D} when f = 2. The example in Table 1a) considers Tempo f = 2. Once process B receives the MPropose with timestamp 6, it bumps its Clock from 6 to 7 and sends a proposal 7 in the MProposeAck. Similarly, processes C and D bump their Clock from 10 to 11 and propose 11. Thus, A receives proposals t_a = 6, t_b = 7, t_c = 11 and t_d = 11, and computes the command’s timestamp as t = max(6, 7, 11) = 11. Since count(11) = 2 ≥ f, the coordinator takes the fast path, even though the proposals did not match. In order to understand why this is safe, assume that the coordinator fails (before sending all the MCommit messages) along with another fast-quorum process. Independently of which ⌈f/2⌉ = 2 fast-quorum processes survive ({B, C} or {B, D} or {C, D}), timestamp 11 is always present and can be recovered as stated by Property 4. This is not the case for the example in Table 1 b). Here A receives t_a = 6, t_b = 7, t_c = 11 and t_d = 6, and again computes t = max(6, 7, 11) = 11. Since count(11) = 1 < f, the coordinator cannot take the fast path: timestamp 11 was proposed solely by C and would be lost if both this process and the coordinator fail. The examples in Table 1 c) and d) consider f = 1, and the fast path is taken in both, independently of

\(^1\)In line 20 we send the message to I_P, even though this set is equal to I_P in the single-partition case. We do this to reuse the pseudocode when presenting the multi-partition protocol in §4.

The timestamps proposed. This is because Tempo fast-path condition count(max{t_j | j ∈ Q}) ≥ f trivially holds with f = 1, and thus Tempo f = 1 always takes the fast path.

Note that when the Clock at a fast-quorum process is below the proposal m sent by the coordinator, i.e., Clock < m, the process makes the same proposal as the coordinator. This is not the case when Clock ≥ m, which can happen when commands are submitted concurrently to the partition. Nonetheless, Tempo is able to take the fast path in some of these situations, as illustrated in Table 1.

Slow path. When the fast-path condition does not hold, the timestamp computed by the coordinator is not yet guaranteed to be persistent: if the coordinator fails before sending all the MCommit messages, a process taking over its job may compute a different timestamp. To maintain Property 1 in this case, the coordinator first reaches an agreement on the computed timestamp with other processes replicating the same partition. This is implemented using single-decree Flexible Paxos [21]. For each identifier we allocate ballot numbers to processes round-robin, with ballot i reserved for the initial coordinator i and ballots higher than r for processes performing recovery. Every process stores for each identifier id the ballot bal[id] it is currently participating in and the last ballot abal[id] in which it accepted a consensus 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 consensus proposal and ballot i to a slow quorum that includes itself. Following Flexible Paxos, the size of the slow quorum is only f+1, rather than a majority like in classical Paxos. As usual in Paxos, a process accepts an MConsensus message only if its ball[id] is not greater than the ballot in the message (line 27). Then it stores the consensus 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 31), it is sure that its consensus proposal will survive the allowed number of failures f, and it thus broadcasts the proposal in an MCommit message.

3.2 Execution Protocol
A process executes committed commands in the timestamp order. To this end, as required by Property 2, a process executes a command only after its timestamp becomes stable, i.e., all commands with a lower timestamp are known. To detect

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>match</th>
<th>fast path</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) f = 2</td>
<td>6</td>
<td>6 → 7</td>
<td>10 → 11</td>
<td>10 → 11</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>b) f = 2</td>
<td>6</td>
<td>6 → 7</td>
<td>10 → 11</td>
<td>5 → 6</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>c) f = 1</td>
<td>6</td>
<td>6 → 7</td>
<td>10 → 11</td>
<td>5 → 6</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>d) f = 1</td>
<td>6</td>
<td>5 → 6</td>
<td>1 → 6</td>
<td>5 → 6</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 1. Tempo examples with r = 5 processes while tolerating f faults. Only 4 processes are depicted, A, B, C and D, with A always acting as the coordinator.
Algorithm 2: Execution protocol at process $i \in I_p$.

\begin{algorithm}
\begin{algorithmic}
\State \textbf{periodically}
\State \hspace{1em} send $\text{MPromises}(\text{Detached, Attached})$ to $I_p$
\State $C \leftarrow \bigcup \{a \cup \{id, a\} \in A \wedge id \in \text{commit} \cup \text{execute}\}$
\State Promises $\leftarrow \text{Promises} \cup D \cup C$
\State $h \leftarrow \text{sort}(\text{highest\_contiguous\_promise}(j) \mid j \in I_p)$
\State $\text{ids} \leftarrow \{id \in \text{commit} \cup \text{execute} \mid ts[id] \leq h[\lceil \frac{t}{2} \rceil]\}$
\State \hspace{1em} for $id \in \text{ids}$ \textbf{ordered by} $\langle \text{ts}[id], id \rangle$
\State \hspace{2em} execute$_p(cmd[id])$; phase[id] $\leftarrow$ \text{EXECUTE}
\State \hspace{1em} \text{highest\_contiguous\_promise}(j)
\State \hspace{2em} $\max\{c \in \mathbb{N} \mid \forall u \in \{1 \ldots c\} \cdot (j, u) \in \text{Promises}\}$
\end{algorithmic}
\end{algorithm}

stability, Tempo tracks which timestamp ranges have been used up by each process using the following mechanism.

**Promise collection.** A promise is a pair $(j, u) \subseteq I_p \times \mathbb{N}$ where $j$ is a process and $u$ a timestamp. Promises can be attached to some command or detached. A promise $(j, u)$ attached to command $c$ means that process $j$ proposed timestamp $u$ for command $c$, and thus will not use this timestamp again. A detached promise $(j, u)$ means that process $j$ will never propose timestamp $u$ for any command.

The function `proposal` is responsible for collecting the promises issued when computing a timestamp proposal $t$ (line 34). This function generates a single attached promise for the proposal $t$, stored in a mapping `Attached` (line 37). The function also generates detached promises for the timestamps ranging from Clock $+1$ up to $t$ (line 36): since the process bumps the Clock to $t$ (line 38), it will never assign a timestamp in this range. Detached promises are accumulated in the `Detached` set. In Table 1 $d_i$, process $B$ generates an attached promise $(B, 6)$, while $C$ generates $(C, 6)$. Process $B$ does not issue detached promises, since its Clock is bumped only by 1, from 5 to 6. However, process $C$ bumps its Clock by 5, from 1 to 6, generating four detached promises: $(C, 2)$, $(C, 3)$, $(C, 4)$, $(C, 5)$.

Algorithm 2 specifies the Tempo execution protocol at a process replicating a partition $p$. Periodically, each process broadcasts its detached and attached promises to the other processes replicating the same partition by sending them in an MPromises message (line 45)\(^2\). When a process receives the promises (line 46), it adds them to a set Promises. Detached promises are added immediately. An attached promise associated with a command identifier $id$ is only added once $id$ is committed or executed (line 47).

**Stability detection.** Tempo determines when a timestamp is stable (Property 2) according to the following theorem.

**Theorem 1.** A timestamp $s$ is stable at a process $i$ if the variable Promises contains all the promises up to $s$ by some set of processes $Q$ with $|Q| \geq \lceil \frac{s}{2} \rceil + 1$.

**Proof.** Assume that at some time $\tau$ the variable Promises at a process $i$ contains all the promises up to $s$ by some set of processes $Q$ with $|Q| \geq \lceil \frac{s}{2} \rceil + 1$. Assume further that a command $c$ with identifier $id$ is eventually committed at timestamp $t \leq s$ at some process $j$, i.e., $j$ receives an MCommit$(id, t)$. We need to show that command $c$ is committed at $i$ at time $\tau$. By Property 3 we have $t = \max\{t_k \mid k \in Q'\}$, where $|Q'| \geq \lceil \frac{s}{2} \rceil + 1$ and $t_k$ is the output of function `proposal(id, _)` at a process $k$. As $Q$ and $Q'$ are majorities, there exists some process $l \in Q \cap Q'$. Then this process attaches a promise $(l, t_l)$ to $c$ (line 37) and $t_l \leq t \leq s$. Since the variable Promises at process $i$ contains all the promises up to $s$ by process $l$, it also contains the promise $(l, t_l)$. According to line 47, when this promise is incorporated into Promises, command $c$ has been already committed at $i$, as required.

A process periodically computes the highest contiguous promise for each process replicating the same partition, and stores these promises in a sorted array $h$ (line 50). It determines the highest stable timestamp according to Theorem 1 as the one at index $\lceil \frac{s}{2} \rceil$ in $h$. The process then selects all the committed commands with a timestamp no higher than the stable one and executes them in the timestamp order, breaking ties using their identifiers. After a command is executed, it is moved to the `execute` phase, which ends its journey.

To gain more intuition about the above mechanism, consider Figure 2, where $r = 3$. There we represent the variable Promises of some process as a table, with processes as columns and timestamps as rows. For example, a promise $(A, 2)$ is in Promises if it is present in column $A$, row 2. There are three sets of promises, $X$, $Y$ and $Z$, to be added to Promises. For each combination of these sets, the right hand side of Figure 2 shows the highest stable timestamp if all the promises in the combination are in Promises. For instance, assume that Promises $= X \cup Z$, so that the set contains promise 2 by $A$, all promises up to 3 by $B$, and all promises up to 2 by $C$. As Promises contains all promises up to 2 by the majority $(B, C)$, timestamp 2 is stable: any uncommitted command $c$ must be committed with a timestamp higher than 2. Indeed, since $c$ is not yet committed, Promises does not contain any promise attached to $c$ (line 47). Moreover, to get committed, $c$ must generate attached promises at a majority of processes (Property 3), and thus, at either $B$ or $C$. If $c$ generates an attached promise at $B$, its coordinator will receive at least proposal 4 from $B$; if at $C$, its coordinator will receive at least proposal 3. In either case, and since the committed timestamp is the highest timestamp proposal, the committed timestamp of $c$ must be at least $3 > 2$, as required.

In our implementation, promises generated by fast-quorum processes when computing their proposal for a command (line 34) are piggybacked on the MProposeAck mes-
sage, and then broadcast by the coordinator in the MCommit message (omitted from the pseudocode). This speeds up stability detection and often allows a timestamp of a command to become stable immediately after it is decided. Notice that when committing a command, Tempo generates detached promises up to the timestamp of that command (line 25). This helps ensuring the liveness of the execution mechanism, since the propagation of these promises contributes to advancing the highest stable timestamp.

### 3.3 Timestamp Stability vs Explicit Dependencies

Prior leaderless protocols \[1, 5, 14, 32, 44\] commit each command \(c\) with a set of explicit dependencies \(\text{dep}[c]\). In contrast, Tempo does not track explicit dependencies, but uses timestamp stability to decide when to execute a command. This allows Tempo to ensure progress under synchrony. Protocols using explicit dependencies do not offer such a guarantee, as they can arbitrarily delay the execution of a command. In practice, this translates into a high tail latency.

Figure 3 illustrates this issue using four commands \(w, x, y, z\) and \(r = 3\) processes. Process A submits \(w\) and \(x\), B submits \(y\), and C submits \(z\). Commands arrive at the processes in the following order: \(w, x, z\) at A; \(y, w\) at B; and \(y, y, z\) at C. Because in this example only process A has seen command \(x\), this command is not yet committed. In Tempo, the above command arrival order generates the following attached promises: \(\{(A, 1), (B, 2)\}\) for \(w\), \(\{(A, 2)\}\) for \(x\), \(\{(B, 1), (C, 2)\}\) for \(y\), and \(\{(C, 1), (A, 3)\}\) for \(z\). Commands \(w, y\), and \(z\) are then committed with the following timestamps: \(ts[w] = 2\), \(ts[y] = 2\), and \(ts[z] = 3\). On the left of Figure 3 we present the Promises variable of some process once it receives the promises attached to the three committed commands. Given these promises, timestamp 2 is stable at the process. Even though command \(x\) is not committed, timestamp stability ensures that its timestamp must be greater than 2. Thus, commands \(w, y\), and \(z\), committed with timestamp 2, can be safely executed. We now show how two approaches that use explicit dependencies behave in the above example.

#### Dependency-based ordering

EPaxos \([32]\) and follow-ups \([5, 14, 44]\) order commands based on their committed dependencies. For example, in EPaxos, the above command arrival order results in commands \(w, y\), and \(z\) committed with the following dependencies: \(\text{dep}[w] = \{y\}\), \(\text{dep}[y] = \{z\}\), \(\text{dep}[z] = \{w, x\}\). These forms the graph shown on the top right of Figure 3. Since the dependency graph may be cyclic (as in Figure 3), commands cannot be simply executed in the order dictated by the graph. Instead, the protocol waits until it forms strongly connected components of the graph and then executes these components one at a time. As we show in [13, §D], the size of such components is a priori unbounded. This can lead to pathological scenarios where the protocol continuously commits commands but can never execute them, even under a synchronous network \([32, 37]\).

It may also significantly delay the execution of committed commands, as illustrated by our example: since command \(x\) has not yet been committed, and the strongly connected component formed by the committed commands \(w, y\), and \(z\) depends on \(x\), no command can be executed – unlike in Tempo. As we demonstrate in our experiments (§6), execution delays in such situations lead to high tail latencies.

#### Dependency-based stability

Caesar \([1]\) associates each command \(c\) not only with a set of dependencies \(\text{dep}[c]\), but also with a unique timestamp \(ts[c]\). Commands are executed in timestamp order, and dependencies are used to determine when a timestamp is stable, and thus when the command can be executed. For this, dependencies have to be consistent with timestamps in the following sense: for any two commands \(c\) and \(c'\), if \(ts[c] < ts[c']\), then \(c \in \text{dep}[c']\). Then the timestamp of a command can be considered stable when the transitive dependencies of the command are committed.

Caesar determines the predecessors of a command while agreeing on its timestamp. To this end, the coordinator of a command sends the command to a quorum together with a timestamp proposal. The proposal is committed when enough processes vote for it. Assume that in our example A proposes \(w\) and \(x\) with timestamps 1 and 4, respectively, B proposes \(y\) with 2, and C proposes \(z\) with 3. When B receives command \(w\) with timestamp proposal 1, it has already proposed \(y\) with timestamp 2. If these proposals succeed and are committed, the above invariant is maintained only if \(w\) is a dependency of \(y\). However, because \(y\) has not yet been committed, its dependencies are unknown and thus B cannot yet ensure that \(w\) is a dependency of \(y\). For this reason, B must block its response about \(w\) until \(y\) is committed. Similarly, command \(y\) is blocked at C waiting for \(z\), and \(z\) is blocked at A waiting for \(x\). This situation, depicted in the bottom right of Figure 3, results in no command being committed – again, unlike in Tempo. In fact, as we show in [13, §D], the blocking
mechanism of Caesar allows pathological scenarios where commands are never committed at all. Similarly to EPaxos, in practice this leads to high tail latencies (§6). In contrast to Caesar, Tempo computes the predecessors of a command separately from agreeing on its timestamp, via background stability detection. This obviates the need for artificial delays in agreement, allowing Tempo to offer low tail latency (§6).

Limitations of timestamp stability. Protocols that track explicit dependencies are able to distinguish between read and write commands. In these protocols writes depend on both reads and writes, but reads only have to depend on writes. The latter feature improves the performance in read-dominated workloads. In contrast, Tempo does not distinguish between read and write commands, so that its performance is not affected by the ratio of reads in the workload. We show in §6 that this limitation does not prevent Tempo from providing similar throughput as the best-case scenario (i.e., a read-only workload) of protocols such as EPaxos and Janus. Adapting techniques that exploit the distinction between reads and writes is left as future work.

4 Multi-Partition Protocol

Algorithm 3 extends the Tempo commit and execution protocols to handle commands that access multiple partitions. This is achieved by submitting a multi-partition command at each of the partitions it accesses using Algorithm 1. Once committed with some timestamp at each of these partitions, the command’s final timestamp is computed as the maximum of the committed timestamps. A command is executed once it is stable at all the partitions it accesses. As previously, commands are executed in the timestamp order.

In more detail, when a process \( i \) submits a multi-partition command \( c \) on behalf of a client (line 1), it sends an MSubmit message to a set \( \{I\}_i \). For each partition \( p \) accessed by \( c \), the set \( \{I\}_i \) contains a responsive replica of \( p \) close to \( i \) (e.g., located in the same data center). The processes in \( \{I\}_i \) then serve as coordinators of \( c \) in the respective partitions, following the steps in Algorithm 1. This algorithm ends with the coordinator in each partition sending an MCommit message to \( \{I\}_c \), i.e., all processes that replicate a partition accessed by \( c \) (lines 20 and 33; note that \( \{I\}_c \neq \{I\}_p \) because \( c \) accesses multiple partitions). Hence, each process in \( \{I\}_c \) receives as many MCommit as the number of partitions accessed by \( c \). Once this happens, the process executes the handler at line 56 in Algorithm 3, which replaces the previous MCommit handler in Algorithm 1. The process computes the final timestamp of the multi-partition command as the highest of the timestamps committed at each partition, moves the command to the commit phase and bumps the Clock to the computed timestamp, generating detached promises.

Commands are executed using the handler at line 60, which replaces that at line 49. This detects command stability using Theorem 1, which also holds in the multi-partition case. The handler signals that a command \( c \) is stable at a partition by sending an MStable message (line 64). Once such a message is received from all the partitions accessed by \( c \), the command is executed. The exchange of MStable messages follows the approach in [4] and ensures and the real-time order constraint in the Ordering property of PSMR (§2).

Example. Figure 4 shows an example of Tempo \( f = 1 \) with \( r = 5 \) and 2 partitions. Only 3 processes per partition are depicted. Partition 0 is replicated at A, B and C, and partition 1 at F, G and H. Processes with the same color (e.g., B and G) are located nearby each other (e.g., in the same machine or data center). Process A and F are the coordinators for some command that accesses the two partitions. At partition 0, A computes 6 as its timestamp proposal and sends it in an MPropose message to the fast quorum \( \{A, B, C\} \) (the downward arrows in Figure 4). These processes make the same proposal, and thus the command is committed at partition 0 with timestamp 6. Similarly, at partition 1, F computes 10 as its proposal and sends it to \( \{F, G, H\} \), all of which propose the same. The command is thus committed at partition 1 with timestamp 10. The final timestamp of the command is then computed as \( \max\{6, 10\} = 10 \).

Assume that the stable timestamp at A is 5 and at F is 9 when they compute the final timestamp for the command. Once F receives the attached promises by the majority \( \{F, G, H\} \), timestamp 10 becomes stable at F. This is not the case at A, as the attached promises by the majority \( \{A, B, C\} \) only make timestamp 6 stable. However, processes A, B and C also generate detached promises up to timestamp 10 when receiving the MCommit messages for the command (line 59). When A receives these promises, it declares timestamp 10 stable. This occurs after two extra message delays: an MCommit

---

**Algorithm 3:** Multi-partition protocol at process \( i \in \{I\}_p \).

```plaintext
56 receive MCommit(id, t_j) from j \in \{I\}_cmd[id]
57 pre: id \in pending
58 ts[id] ← max\{t_j | j \in P\}; phase[id] ← COMMIT
59 bump(ts[id])
60 periodically
61 h ← sort\{highest_contiguous_promise(j) | j \in \{I\}_p\}
62 ids ← \{id \in commit | ts[id] ≤ h[|I|])
63 for id ∈ ids ordered by \{ts[id], id\}
64 send MStable(id) to \{I\}_cmd[id]
65 wait receive MStable(id) from \forall j \in \{I\}_cmd[id]
66 execute_p(cmd[id]); phase[id] ← execute
67 receive MPropose(id, c, Q, t) from j
68 send MBump(id, ts[id]) to \{I\}_c
69 receive MBump(id, t)
70 pre: id ∈ propose
71 bump(t)
```
Algorithm 1. It then additionally sends an
and
A
10
Janus [33]. With a genuine protocol, partitioning the ap-
partitions are colocated in the same machine, the message
increase in the number of partitions (and thereby of avail-
Genuineness and parallelism. Faster stability. Tempo avoids the above extra delays by
generating the detached promises needed for stability ear-
erlier than in the MCommit handler. For this we amend the
MPropose handler as shown in Algorithm 3. When a process
receives an MPropose message, it follows the same steps as in
Algorithm 1. It then additionally sends an MBump message
containing its proposal to the nearby processes that replicate a
partition accessed by the command (line 68). Upon receiv-
ing this message (line 69), a process bumps its Clock to the
timestamp in the message, generating detached promises.

In Figure 4, MBump messages are depicted by horizontal
dashed arrows. When G computes its proposal 10, it sends an
MBump message containing 10 to process B. Upon recep-
tion, B bumps its Clock to 10, generating detached promises
up to that value. The same happens at A and C. Once the
detached promises by the majority \{A, B, C\} are known at A,
the process again declares 10 stable. In this case, A receives
the required detached promises in two message delays earlier
than when these promises are generated via MCommit. This
strategy often reduces the number of message delays neces-
sary to execute a multi-partition command. However, it is not
always sufficient (e.g., imagine that H proposed 11 instead of
10), and thus, the promises issued in the MCommit handler
(line 59) are still necessary for multi-partition commands.

Genuineness and parallelism. The above protocol is genu-
une: for every command c, only the processes in \(\mathcal{I}_c\) take steps to order and execute c [17]. This is not the case for
existing leaderless protocols for partial replication, such as
Janus [33]. With a genuine protocol, partitioning the ap-
lication state brings scalability in parallel workloads: an
increase in the number of partitions (and thereby of avail-
able machines) leads to an increase in throughput. When
partitions are colocated in the same machine, the message
passing in Algorithm 3 can be optimized and replaced by

shared-memory operations. Since Tempo runs an indepen-
dent instance of the protocol for each partition replicated at
the process, the resulting protocol is highly parallel.

5 Recovery Protocol

The initial coordinator of a command at some partition \(p\)
may fail or be slow to respond, in which case Tempo allows a
process to take over its role and recover the command’s
timestamp. We now describe the protocol Tempo follows in
this case, which is inspired by that of Atlas [14]. This
protocol at a process \(i \in \mathcal{I}_p\) is given in Algorithm 4. We
use initial\(_p\)(id) to denote a function that extracts from the
command identifier id its initial coordinator at partition \(p\).

A process takes over as the coordinator for some com-
mand with identifier id by calling function recover(id) at
line 72. Only a process with \(id \in \text{pending}\) can take over as a
coordinator (line 73): this ensures that the process knows the
command payload and fast quorums. In order to find out if a decision on the timestamp 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 74) and sends an MRec
message with this ballot to all processes.

As is standard in Paxos, a process accepts an MRec
message only if the ballot in the message is greater than its
bal[id] (line 77). If bal[id] is still 0 (line 78), the process
checks the command’s phase to decide if it should compute
its timestamp proposal for the command. If phase[id] =
payload (line 79), the process has not yet computed a time-
stamp proposal, and thus it does so at line 80. It also sets the
cmd:’s phase to recover-r, which records that the
timestamp proposal was computed in the MRec handler. Other-
wise, if phase[id] = propose (line 82), the process has al-
ready computed a timestamp proposal at line 15. In this case,
the process simply sets the cmd:’s phase to recover-p,
which records that the timestamp proposal was computed in
the MPropose handler. Finally, the process sets bal[id] to
the new ballot and replies with an MRecAck message con-
taining the timestamp (ts), the command’s phase (phase) and
the ballot at which the timestamp was previously accepted in
consensus (abal). Note that abal[id] = 0 if the process
has not yet accepted any consensus proposal. Also note that
lines 79 and 82 are exhaustive: these are the only possible
phases when id \(\in\) pending (line 77) and bal[id] = 0 (line 78),
as recovery phases have non-zero ballots (line 84).

In the MRecAck handler (line 86), the new coordinator
computes the cmd:’s timestamp given the information in the
MRecAck messages and sends it in an MConsensus
message to all processes. As in Flexible Paxos, the new co-
ordinator waits for \(r - f\) such messages. This guarantees
that, if a quorum of \(f + 1\) processes accepted an MConsensus
message with a timestamp (which could have thus been sent
in an MCommit message), the new coordinator will find out

![Figure 4. Example of Tempo with 2 partitions. Next to each process we show the clock updates upon receiving MPropose messages and, in dashed boxes, the updates upon receiving MCommit or MBump messages (whichever occurs first).](image)
Algorithm 4: Recovery protocol at process \( i \in I_p \).

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>\textbf{recover}(id)</td>
</tr>
<tr>
<td>73</td>
<td>\textbf{pre}: id \in \textbf{pending}</td>
</tr>
<tr>
<td>74</td>
<td>( b \leftarrow i + r(\frac{\text{ball}[id] - 1}{r} + 1) )</td>
</tr>
<tr>
<td>75</td>
<td>\textbf{send} MRec(id, b) to ( I_p )</td>
</tr>
<tr>
<td>76</td>
<td>\textbf{receive} MRec(id, b) from ( j )</td>
</tr>
<tr>
<td>77</td>
<td>\textbf{pre}: id \in \textbf{pending} \land \text{ball}[id] &lt; b</td>
</tr>
<tr>
<td>78</td>
<td>if \text{ball}[id] = 0 then</td>
</tr>
<tr>
<td>79</td>
<td>if phase[i] = \textbf{PAYLOAD} then</td>
</tr>
<tr>
<td>80</td>
<td>ts[i] \leftarrow \text{proposal}(id, 0)</td>
</tr>
<tr>
<td>81</td>
<td>phase[i] \leftarrow \textbf{RECOVER-R}</td>
</tr>
<tr>
<td>82</td>
<td>else if phase[i] = \textbf{PROPOSE} then</td>
</tr>
<tr>
<td>83</td>
<td>phase[i] \leftarrow \textbf{RECOVER-P}</td>
</tr>
<tr>
<td>84</td>
<td>\textbf{ball}[id] \leftarrow b</td>
</tr>
<tr>
<td>85</td>
<td>\textbf{send} MRecAck(id, ts[i], phase[i], abal[id], b) to ( j )</td>
</tr>
<tr>
<td>86</td>
<td>\textbf{receive} MRecAck(id, t, ph, id, b) from ( \forall j \in Q )</td>
</tr>
<tr>
<td>87</td>
<td>\textbf{pre}: bal[id] = b \land</td>
</tr>
<tr>
<td>88</td>
<td>if \exists k \in \text{Quorums}[id][p] then</td>
</tr>
<tr>
<td>89</td>
<td>let ( k ) be such that ( abk ) is maximal</td>
</tr>
<tr>
<td>90</td>
<td>\textbf{send} MConsensus(id, ( t_k ), b) to ( I_p )</td>
</tr>
<tr>
<td>91</td>
<td>else</td>
</tr>
<tr>
<td>92</td>
<td>( I \leftarrow Q \cap \text{quorums}[id][p] )</td>
</tr>
<tr>
<td>93</td>
<td>( s \leftarrow \text{initial}_p(id) \in I \lor \exists k \in I \cdot ph_k = \text{RECOVER-P} )</td>
</tr>
<tr>
<td>94</td>
<td>( Q' \leftarrow \text{if } s \text{ then } Q \text{ else } I )</td>
</tr>
<tr>
<td>95</td>
<td>( t \leftarrow \max{t_j \mid j \in Q'} )</td>
</tr>
<tr>
<td>96</td>
<td>\textbf{send} MConsensus(id, t, b) to ( I_p )</td>
</tr>
</tbody>
</table>

about this timestamp. To maintain Property 1, if any process previously acceptable a consensus proposal (line 88), by the standard Paxos rules [21, 29], the coordinator selects the proposal accepted at the highest ballot (line 89).

If no consensus proposal has been accepted before, the new coordinator first computes at line 92 the set of processes \( I \) that belong both to the recovery quorum \( Q \) and the fast quorum quorums \([id][p] \). Then, depending on whether the initial coordinator replied and in which handler the processes in \( I \) have computed their timestamp proposal, there are two possible cases that we describe next.

1) The initial coordinator replies or some process in \( I \) has computed its timestamp proposal in the MRec handler \((s = \text{true}, \text{line 93})\). In either of these two cases the initial coordinator could not have taken the fast. If the initial coordinator replies \((\text{initial}_p(id) \in I)\), then it has not taken the fast path before receiving the MRec message from the new one, as it would have \( id \in \text{commit} \cup \text{execute} \) and the MRec precondition requires \( id \in \text{pending} \) (line 77). It will also not take the fast path in the future, since when processing the MRec message it sets the command’s phase to \text{RECOVER-P} (line 83), which invalidates the MProposeAck precondition (line 18). On the other hand, even if the initial coordinator replies but some fast-quorum process in \( I \) has computed its timestamp proposal in the MRec handler, the fast path will not be taken either. This is because the command’s phase at such a process is set to \text{RECOVER-R} (line 81), which invalidates the MPropose precondition (line 13). Then, since the MProposeAck precondition requires a reply from all fast-quorum processes, the initial coordinator will not take the fast path. Thus, in either case, the initial coordinator never takes the fast path. For this reason, the new coordinator can choose the command’s timestamp in any way, as long as it maintains Property 3. Since \(|Q| = r - f \geq r - \lfloor \frac{n - 1}{f} \rfloor \geq \lfloor \frac{r}{f} \rfloor + 1\), the new coordinator has the output of \text{proposals} by a majority of processes, and thus it computes the command’s timestamp with max (line 95), respecting Property 3.

2) The initial coordinator does not reply and all processes in \( I \) have computed their timestamp proposal in the MPropose handler \((s = \text{false}, \text{line 93})\). In this case the initial coordinator could have taken the fast path with some timestamp \( t = \max\{t_j \mid j \in \text{quorums}[id][p]\} \) and, if it did, the new coordinator must choose that same timestamp \( t \). Given that the recovery quorum \( Q \) has size \( r - f \) and the fast quorum \( \text{quorums}[id][p] \) has size \( \lfloor \frac{r}{f} \rfloor + f \), the set of processes \( I = Q \cap \text{quorums}[id][p] \) contains at least \( \lfloor \frac{r}{f} \rfloor \) processes (distinct from the initial coordinator, as it did not reply). Furthermore, recall that the processes from \( I \) have the command’s phase set to \text{RECOVER-P} (line 83), which invalidates the MPropose precondition (line 13). Hence, if the initial coordinator took the fast path, then each process in \( I \) must have processed its MPropose before the MRec of the new coordinator, and reported in the latter the timestamp from the former. Then using Property 4, the new coordinator recovers \( t \) by selecting the highest timestamp reported in \( I \) (line 95).

Additional liveness mechanisms. As is standard, to ensure the progress of recovery, \textsc{Tempo} nominates a single process to call \texttt{recover} using a partition-wide failure detector [6], and ensures that this process picks a high enough ballot. \textsc{Tempo} additionally includes a mechanism to ensure that, if a correct process receives an MPayload or an MCommit message, then all correct process do; this is also necessary for recovery to make progress. For brevity, we defer a detailed description of these mechanisms to [13, §B].

Correctness. We have rigorously proved that \textsc{Tempo} satisfies the PSMR specification (§2), even in case of failures. Due to space constraints, we defer the proof to [13, §C].

6 Performance Evaluation

In this section we experimentally evaluate \textsc{Tempo} in deployments with full replication (i.e., each partition is replicated at all processes) and partial replication. We compare \textsc{Tempo} with Flexible Paxos (FPaxos) [21], EPaxos [32], Atlas [14], Caesar [1] and Janus [33]. FPaxos is a variant of Paxos that, like \textsc{Tempo}, allows selecting the allowed number of failures \( f \) separately from the replication factor \( r \); it uses quorums of size \( f + 1 \) during normal operation and quorums of size \( r - f \).
during recovery. EPaxos, Atlas and Caesar are leaderless protocols that track explicit dependencies (§3.3). EPaxos and Caesar use fast quorums of size $[\frac{r}{2^2}]$ and $[\frac{r}{2^3}]$, respectively. Atlas uses fast quorums of the same size as TEMPO, i.e., $[\frac{r}{2^2}] + f$. Atlas also improves the condition EPaxos uses for taking the fast path: e.g., when $r = 5$ and $f = 1$, Atlas always processes commands via the fast path, unlike EPaxos. To avoid clutter, we exclude the results for EPaxos from most of our plots since its performance is similar to (but never better than) Atlas $f = 1$. Janus is a leaderless protocol that generalizes EPaxos to the setting of partial replication. It is based on an unoptimized version of EPaxos whose fast quorums contain all replicas in a given partition. Our implementation of Janus is instead based on Atlas, which yields quorums of the same size as TEMPO and a more permissive fast-path condition. We call this improved version Janus*.

This protocol is representative of the state-of-the-art for partial replication, and the authors of Janus have already compared it extensively to prior approaches (including MDCC [26], Tapir [46] and 2PC over Paxos [8]).

6.1 Implementation

To improve the fairness of our comparison, all protocols are implemented in the same framework which consists of 33K lines of Rust and contains common functionality necessary to implement and evaluate the protocols. This includes a networking layer, an in-memory key-value store, dstat monitoring, and a set of benchmarks (e.g. YCSB [7]). The source code of the framework is available at github.com/vitorenesduarte/fantoch.

The framework provides three execution modes: cloud, cluster and simulator. In the cloud mode, the protocols run in wide area on Amazon EC2. In the cluster mode, the protocols run in a local-area network, with delays injected between the machines to emulate wide-area latencies. Finally, the simulator runs on a single machine and computes the observed client latency in a given wide-area configuration when CPU and network bottlenecks are disregarded. Thus, the output of the simulator represents the best-case latency for a given scenario. Together with dstat measurements, the simulator allows us to determine if the latencies obtained in the cloud or cluster modes represent the best-case scenario for a given protocol or are the effect of some bottleneck.

6.2 Experimental Setup

Testbeds. As our first testbed we use Amazon EC2 with c5.2xlarge instances (machines with 8 virtual CPUs and 16GB of RAM). Experiments span up to 5 EC2 regions, which we call sites: Ireland (eu-west-1), Northern California (us-west-1), Singapore (ap-southeast-1), Canada (ca-central-1), and São Paulo (sa-east-1). The average ping latencies between these sites range from 72ms to 338ms; we defer precise numbers to [13, §A]. Our second testbed is a local cluster where we inject wide-area delays similar to those observed in EC2.

The cluster contains machines with 6 physical cores and 32GB of RAM connected by a 10GBit network.

Benchmarks. We first evaluate full replication deployments (§6.3) using a microbenchmark where each command carries a key of 8 bytes and (unless specified otherwise) a payload of 100 bytes. Commands access the critical path when they carry the same key, in which case we say that they conflict. To measure performance under a conflict rate $\rho$ of commands, a client chooses key 0 with a probability $\rho$, and some unique key otherwise. We next evaluate partial replication deployments (§6.4) using YCSB+T [10], a transactional version of the YCSB benchmark [7]. Clients are closed-loop and always deployed in separate machines located in the same regions as servers. Machines are connected via 16 TCP sockets, each with a 16MB buffer. Sockets are flushed every 5ms or when the buffer is filled, whichever is earlier.

6.3 Full Replication Deployment

Fairness. We first evaluate a key benefit of leaderless SMR, its fairness: the fairer the protocol, the more uniformly it satisfies different sites. We compare TEMPO, Atlas and FPaxos when the protocols are deployed over 5 EC2 sites under two fault-tolerance levels: $f \in \{1, 2\}$. We also compare with Caesar which tolerates $f = 2$ failures in this setting. At each site we deploy 512 clients that issue commands with a low conflict rate ($2\%$).

Figure 5 depicts the per-site latency provided by each protocol. The FPaxos leader site is Ireland, as we have determined that this site produces the fairest latencies. However, even with this leader placement, FPaxos remains significantly unfair. When $f = 1$, the latency observed by clients at the leader site is 82ms, while in São Paulo and Singapore it is 267ms and 264ms, respectively. When $f = 2$, the clients in Ireland, São Paulo and Singapore observe respectively the latency of 142ms, 325ms and 323ms. Overall, the performance at non-leader sites is up to 3.3x worse than at the leader site.

Due to their leaderless nature, TEMPO, Atlas and Caesar satisfy the clients much more uniformly. With $f = 1$, TEMPO and Atlas offer similar average latency – 138ms for TEMPO and 155ms for Atlas. However, with $f = 2$ TEMPO clearly outperforms Atlas – 178ms versus 257ms. Both protocols use fast quorums of size $[\frac{r}{2^2}] + f$. But because quorums for $f = 2$ are larger than for $f = 1$, the size of the dependency sets in Atlas increases. This in turn increases the size of the strongly connected components in execution (§3.3). Larger components result in higher average latencies, as reported in Figure 5. Caesar provides the average latency of 195ms, which is 17ms higher than TEMPO $f = 2$. Although Caesar and TEMPO $f = 2$ have the same quorum size with $r = 5$, the blocking mechanism of Caesar delays commands in the critical path (§3.3), resulting in slightly higher average latencies. As we now demonstrate, both Caesar and Atlas have much higher tail latencies than TEMPO.
Tail latency. Figure 6 shows the latency distribution of various protocols from the 95th to the 99.99th percentiles. At the top we give results with 256 clients per site, and at the bottom with 512, i.e., the same load as in Figure 5.

The tail of the latency distribution in Atlas, EPaxos and Caesar is very long. It also sharply deteriorates when the load increases from 256 to 512 clients per site. For Atlas \( f = 1 \), the 99th percentile increases from 385ms to 586ms while the 99.9th percentile increases from 1.3s to 2.4s. The trend is similar for Atlas \( f = 2 \), making the 99.9th percentile increase from 4.5s to 8s. The performance of EPaxos lies in between Atlas \( f = 1 \) and Atlas \( f = 2 \). This is because with 5 sites EPaxos has the same fast quorum size as Atlantic \( f = 1 \), but takes the slow path with a similar frequency to Atlas \( f = 2 \). For Caesar, increasing the number of clients also increases the 99th percentile from 893ms to 991ms and 99.9th percentile from 1.6s to 2.4s. Overall, the tail latency of Atlas, EPaxos and Caesar reaches several seconds, making them impractical in these settings. These high tail latencies are caused by ordering commands using explicit dependencies, which can arbitrarily delay command execution (§3.3).

In contrast, TEMPO provides low tail latency and predictable performance in both scenarios. When \( f = 1 \), the 99th, 99.9th and 99.99th percentiles are respectively 280ms, 361ms and 386ms (averaged over the two scenarios). When \( f = 2 \), these values are 449ms, 552ms and 562ms. This represents an improvement of 1.4-8x over Atlas, EPaxos and Caesar with 256 clients per site, and an improvement of 4.3-14x with 512. The tail of the distribution is much shorter with TEMPO due to its efficient execution mechanism, which uses timestamp stability instead of explicit dependencies.

We have also run the above scenarios in our wide-area simulator. In this case the latencies for Atlas, EPaxos and Caesar are up to 30% lower, since CPU time is not accounted for. The trend, however, is similar. This confirms that the latencies reported in Figure 6 accurately capture the effect of long dependency chains and are not due to a bottleneck in the execution mechanism of the protocols.

Increasing load and contention. We now evaluate the performance of the protocols when both the client load and contention increases. This experiment, reported in Figure 7, runs over 5 sites. It employs a growing number of clients per site (from 32 to 20K), where each client submits commands with a payload of 4KB. The top scenario of Figure 7 uses the same conflict rate as in the previous experiments (2%), while the bottom one uses a moderate conflict rate of 10%. The heatmap shows the hardware utilization (CPU, inbound and outbound network bandwidth) for the case when the conflict rate is 2%. For leaderless protocols, we measure the hardware utilization averaged across all sites, whereas for EPaxos, we only show this measure at the leader site. The experiment runs on a local cluster with emulated wide-area latencies, to have a full control over the hardware.

As seen in Figure 7, the leader in EPaxos quickly becomes a bottleneck when the load increases since it has to broadcast each command to all the processes. For this reason, EPaxos provides the maximum throughput of only 53K ops/s with \( f = 1 \) and of 45K ops/s with \( f = 2 \). The protocol saturates at around 4K clients per site, when the outgoing network bandwidth reaches 95% usage. The fact that the leader can be a bottleneck in leader-based protocol has been reported by several prior works [14, 4, 25, 32, 44].

FPaxos is not affected by contention and the protocol has identical behavior for the two conflict rates. On the contrary, Atlas performance degrades when contention increases. With a low conflict rate (2%), the protocol provides the maximum throughput of 129K ops/s with \( f = 1 \) and of 127K ops/s with \( f = 2 \). As observed in the heatmap (bottom of Figure 7), Atlas cannot fully leverage the available hardware. CPU usage reaches at most 59%, while network utilization reaches 41%. This low value is due to a bottleneck in the execution mechanism: its implementation, which follows the one by the authors of EPaxos, is single-threaded. Increasing the conflict rate to 10% further decreases hardware utilization: the maximum CPU usage decreases to 40% and network to 27% (omitted from Figure 7). This sharp decrease is due to the dependency chains, whose sizes increase with higher contention, thus requiring fewer clients to bottleneck execution. As a consequence, the throughput of Atlas decreases by 36% with \( f = 1 \) (83K ops/s) and by 48% with
We consider 3 payload sizes: 256B, 1KB and 4KB. The numbers for 4KB with batching disabled correspond to the ones in Figure 7. Because with 4KB and 1KB FPaxos bottlenecks in the network (Figure 7), enabling batching does not help. When the payload size is reduced further to 256B, the bottleneck shifts to the leader thread. In this case, enabling batching allows FPaxos to increase its performance by 4x. Since TEMPO performs heavier computations than FPaxos, the use of batches in TEMPO only brings a moderate improvement: 1.6x with 256B and 1.3x with 1KB. In the worst case, with 4KB, the protocol can even perform less efficiently.

While batching can boost leader-based SMR protocols, the benefits are limited for leaderless ones. However, because leaderless protocols already efficiently balance resource usage across replicas, they can match or even outperform the performance of leader-based protocols, as seen in Figure 8.

### 6.4 Partial Replication Deployment

We now compare TEMPO with Janus∗ using the YCSB+T benchmark. We define a shard as set of several partitions co-located in the same machine. Each partition contains a single YCSB key. Each shard holds 1M keys and is replicated at 3 sites (Ireland, N. California and Singapore) emulated in our cluster. Clients submit commands that access two co-located keys picked at random following the YCSB access pattern (a zipfian distribution). In Figure 9 we show the maximum throughput for both TEMPO and Janus∗ under low (zipf = 0.5) and moderate contention (zipf = 0.7). Three workloads are considered for Janus∗: 0% writes as the best-case scenario, 5% writes and 50% writes.
(w = 50%, YCSB workload A). The read-only workload is a rare workload in SMR deployments. It represents the best-case scenario for Janus*, which we use as a baseline. Since Tempo does not distinguish between reads and writes (§3.3), we have a single workload for this protocol.

Janus* performance is greatly affected by the ratio of writes and by contention. More writes and higher contention translate into larger dependency sets, which bottleneck execution faster. This is aggravated by the fact that Janus* is non-genuine, and thus requires cross-shard messages to order commands. With zipf = 0.5, increasing w from 0% to 5% reduces throughput by 25-26%. Increasing w from 0% to 50% reduces throughput by 49-56%. When contention increases (zipf = 0.7), the above reductions on throughput are larger, reaching 36-60% and 87%-94%, respectively.

Tempo provides nearly the same throughput as the best-case scenario for Janus* (w = 0%). Moreover, its performance is virtually unaffected by the increased contention. This comes from the parallel and genuine execution brought by the use of timestamp stability (§4). Overall, Tempo provides 385K ops/s with 2 shards, 606K ops/s with 4 shards, and 784K ops/s with 6 shards (averaged over the two zipf values). Compared to Janus* w = 5% and Janus* w = 50%, this represents respectively a speedup of 1.2-2.5x and 2-16x.

The tail latency issues demonstrated in Figure 6 also carry over to partial replication. For example, with 6 shards, zipf = 0.7 and w = 5%, the 99.99th percentile for Janus* reaches 1.3s, while Tempo provides 421ms. We also ran the same set of workloads for the full replication case and the speed up of Tempo with respect to EPaxos and Atlas is similar.

7 Related Work

Timestamping (aka sequencing) is widely used in distributed systems. In particular, many storage systems orchestrate data access using a fault-tolerant timestamping service [2, 3, 36, 42, 47], usually implemented by a leader-based SMR protocol [29, 35]. As reported in prior works, the leader is a potential bottleneck and is unfair with respect to client locations [14, 24, 25, 32, 44]. To sidestep these problems, leaderless protocols order commands in a fully decentralized manner. Early protocols in this category, such as Mencius [31], rotated the role of leader among processes. However, this made the system run at the speed of the slowest replica. More recent ones, such as EPaxos [32] and its follow-ups [5, 14, 44], order commands by agreeing on a graph of dependencies (§3.3). Tempo builds on one of these follow-ups, Atlas [14], which leverages the observation that correlated failures in geo-distributed systems are rare [8] to reduce the quorum size in leaderless SMR. As demonstrated by our evaluation (§6.3), dependency-based leaderless protocols exhibit high tail latency and suffer from bottlenecks due to their expensive execution mechanism.

8 Conclusion

We have presented Tempo – a new SMR protocol for geo-distributed systems. Tempo follows a leaderless approach, ordering commands in a fully decentralized manner and thus offering similar quality of service to all clients. In contrast to previous leaderless protocols, Tempo determines the order of command execution solely based on scalar timestamps, and cleanly separates timestamp assignment from detecting timestamp stability. Moreover, this mechanism easily extends to partial replication. As shown in our evaluation, Tempo’s approach enables the protocol to offer low tail latency and high throughput even under contended workloads.

Acknowledgments. We thank our shepherd, Natacha Crooks, as well as Antonios Katsarakis, Ricardo Macedo, Georges Younes for comments and suggestions. We also thank Balaji Arun, Roberto Palmieri and Sebastiano Peluso for discussions about Caesar. Vitor Enes was supported by an FCT PhD Fellowship (PD/BD/142927/2018). Pierre Sutra was 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 AWS Cloud Credit for Research program.
References


