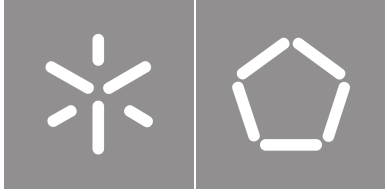
Universidade do Minho
Escola de Engenharia

Vitor Manuel Enes Duarte

Planet-Scale Leaderless Consensus

November, 2022



Universidade do Minho
Escola de Engenharia

Vitor Manuel Enes Duarte

Planet-Scale Leaderless Consensus

Doctorate Thesis
Doctorate in Computer Science
Specialization in Distributed Systems

Work developed under the supervision of:
Alexey Gotsman
Carlos Baquero

November, 2022

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.9.0) [1].

Acknowledgements

Without the guidance of my advisors, Alexey Gotsman and Carlos Baquero, this thesis would not exist. I didn't know where we were going most of the times (maybe we never do), but with your help and support somehow we got here. Without me realizing it at the time, this thesis started during my internship at IMDEA. I am immensely thankful for having worked closely with Alexey since then. His patience, invested time and detailed feedback made me a better researcher and this thesis possible. I still remember the day when I entered Carlos' office to bother him with my interest in CRDTs. Most certainly I would not be here if he hadn't believe in me back then and since. I am tremendously grateful for having worked with Pierre Sutra and Paulo Sérgio Almeida. You were also my advisors, just not on paper. I learned so much from you and I cannot thank you enough.

I especially want to thank Matthieu Perrin, Tuanir França Rezende, Christopher S. Meiklejohn, Junghun Yoo, Peter Van Roy, Annette Bieniusa, João Leitão, and Ali Shoker, with whom I have collaborated during all these years. I am thankful for the numerous anonymous reviewers, as well as Liuba Shrira and Natacha Crooks, all of whom shepherded papers containing results that appear on this thesis. I am also very thankful for Lennart Oldenburg, Antonios Katsarakis, Gonçalo Tomás, Dimitris Vasilas, Manuel Bravo, Borja de Régil, Fedor Ryabinin, Artem Khyzha, Marc Shapiro, Andrey Kuprianov, Adi Seredinschi, and Murat Demirbas for their research advice and comments on papers. I am thankful for everyone at HASLab, IMDEA and Informal Systems for being so welcoming and helpful throughout these years.

I am so grateful for all of my labmates. Georges, Ricardo, Rogério, Cláudia, Fábio, Tânia, Neves, João and Marco, thank you for always being there to listen to my ideas and worries. Georges, my friend, I am immensely grateful for having shared this journey with you since day one, for all our trips and discussions about research and life.

Friends, I am thankful for your love, help, support, presence and companionship, for our conversations, walks, travels, drinks and laughs. Ana, Rita, Inês, Palma, Rafa, Zé, Cabrita, Tatiana, Catarina, Fialho, Rita, Quim, Andreia, Anthony, Salomé, Marta, Marco, Ricardo, Simone, Elisa, Verónica, Filipa, Filipe, Linhares, AP, Carolina, Isac, Hugo, Ribeiro, Frank, Póvoas, Gil, Sniper, Rúben, Scout, Caxinas, Nélson, Bruno, and Rocky, I am tremendously grateful for your friendship.

Above all, a huge thanks to all my family, specially to my mother, father and sister, for always being there for me. A special thanks to my uncle Nuno, for being an inspiration in many aspects of life, to Helena and Rogério, for their trust (and my newly found love for books), and to my godmother, godfather, and cousins.

This work was partially supported by an FCT – “Fundação para a Ciência e Tecnologia” – PhD Fellowship (PD/BD/142927/2018).

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

Resumo

Planet-Scale Leaderless Consensus

As aplicações de web modernas replicam os seus dados à escala planetária e exigem fortes garantias na coerência dos seus dados mais críticos. Essas garantias são geralmente fornecidas por meio de replicação de máquina de estados (RME). Avanços recentes em RME concentraram-se em protocolos sem líder, pois estes melhoram o desempenho e a disponibilidade das soluções tradicionais baseadas em Paxos. Embora os protocolos sem líder se tenham mostrado muito promissores, estes são ainda pouco adequados para sistemas de escala planetária, pois utilizam grandes quóruns, oferecem um desempenho imprevisível e têm mecanismos de recuperação complexos. Nesta tese propomos dois protocolos sem líder, `Atlas` e `Tempo`, adaptados para sistemas de escala planetária. O `Atlas` minimiza o tamanho dos seus quóruns fazendo uso da observação de que falhas simultâneas em centros de dados são raras. Também processa uma percentagem elevada de comandos da aplicação em uma única *round trip*, mesmo quando estes comandos conflituam. O `Atlas` consegue isto com um mecanismo de recuperação que é significativamente mais simples do que os protocolos sem líder que o precederam. O `Tempo` baseia-se no `Atlas`, mas atinge um rendimento superior e oferece um desempenho previsível mesmo em cargas de trabalho com elevado nível de conflitos. Para obter estes benefícios, o `Tempo` marca cada comando da aplicação com uma *timestamp* e executa-o somente após esta timestamp se tornar estável, ou seja, quando todos os comandos com uma timestamp menor são conhecidos. Ambos os mecanismos para gerar uma timestamp e detetar quando esta fica estável são totalmente descentralizados, evitando assim a necessidade de um líder. Avaliámos o `Atlas` e o `Tempo` em ambientes geo-distribuídos reais e simulados e demonstramos que eles superam as alternativas oferecidas pelo estado da arte.

Palavras-chave: Consenso distribuído, Geo-replicação, Tolerância a falhas.

Abstract

Planet-Scale Leaderless Consensus

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 performance and availability of traditional Paxos-based solutions. Although leaderless protocols have shown great promise, they are poorly suited to planet-scale systems as they leverage large quorums, offer unpredictable performance and have complex recovery mechanisms. In this thesis we propose two leaderless protocols, `Atlas` and `Tempo`, tailored to planet-scale systems. `Atlas` minimizes the size of its quorums by making use of the observation that concurrent data center failures are rare. It also processes a high percentage of accesses in a single round trip, even when these conflict. `Atlas` achieves this while having a recovery mechanism that is significantly simpler than that of previous leaderless protocols. `Tempo` builds upon `Atlas`, but 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. We evaluate `Atlas` and `Tempo` in both real and simulated geo-distributed environments and demonstrate that they outperform state-of-the-art alternatives.

Keywords: Consensus, Fault tolerance, Geo-replication.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Modern online applications run at multiple sites scattered across the globe: they are now *planet-scale*. Deploying applications in this way enables high availability and low latency, by allowing clients to access the closest responsive site. A major challenge in developing planet-scale applications is that many of their underlying components, such as coordination kernels [2, 3] and critical databases [4], require strong guarantees about the consistency of replicated data.

The classical way of maintaining strong consistency in a distributed service is *state-machine replication (SMR)* [5]. In SMR, a service is defined by a deterministic state machine, and each site maintains its own local replica of the machine. An *SMR protocol* coordinates the execution of commands at the sites, ensuring that they stay in sync. The resulting system is *linearizable* [6], i.e., it behaves as if commands are executed sequentially by a single site.

Unfortunately, existing SMR protocols are poorly suited to planet-scale systems. Common SMR protocols, such as Paxos [7] and Raft [8], are rooted in cluster computing where a distinguished *leader* site determines the order in which client commands are executed at the replicas. This is unfair to clients far away from the leader. It impairs scalability, since the leader becomes a bottleneck when the load increases. It also harms availability as, if the leader fails, the system cannot serve requests until a new one is elected. Moreover, adding more sites to the system does not help, but on the contrary, hinders performance, requiring the leader to replicate commands to more sites on the critical path.

Recent efforts [9–12] 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. Compared to centralized solutions, leaderless SMR offers lower average latency, fairer latency distribution with respect to client locations, and higher availability. Moreover, leaderless protocols usually exploit the fact that commands in SMR applications frequently commute [3, 4], and for the replicated state machine to be linearizable, it is enough that replicas only agree on the order of non-commuting (aka conflicting) commands [13, 14]. This permits processing a command in one round trip from the closest replica using a *fast path*, e.g., when the command commutes with all commands concurrently submitted for execution. In the presence of concurrent conflicting commands, the protocol may sometimes have to take a *slow path*, which requires two round trips.

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 [15] which adapted a leaderless SMR protocol called Egalitarian Paxos (EPaxos) [11] to the setting of partial replication. The resulting protocol provided better performance than classical solutions such as two-phase commit layered over Paxos.

Although leaderless protocols show great promise, they have not yet seen industry adoption. In order to be practical for planet-scale systems, we argue that leaderless protocols have to provide the following features: *(i) low latency*, *(ii) simple recovery*, and *(iii) predictable performance*. In this thesis we propose `Atlas` and `Tempo`, two leaderless protocols tailored to planet-scale systems that aim to offer these features. `Atlas` provides the first two features (*low latency* and *simple recovery*). `Tempo` builds on `Atlas`, and also offers the third one (*predictable performance*). Next we cover each of these features in detail.

Low latency We observe that common SMR protocols provide a level of fault-tolerance that is unnecessarily high in a geo-distributed setting. These protocols allow any minority of sites to fail simultaneously: e.g., running a typical protocol over 11 data centers would tolerate 5 of them failing. However, natural disasters leading to the loss of a data center are rare, and planned downtime can be handled by reconfiguring the unavailable site out of the system [7, 16]. Furthermore, temporary data center outages (e.g., due to connectivity issues) typically have a short duration [17], and, as we confirm experimentally in our evaluation (§5), rarely happen concurrently. For this reason, industry practitioners assume that the number of concurrent site failures in a geo-distributed system is low, e.g., 1 or 2 [4]. Motivated by this, `Atlas` allows choosing the maximum number of sites that can fail (f) independently of the overall number of sites (r), and is optimized for small values of the former.

Making `Atlas` offer better latency for larger-scale deployments required two key changes to the baseline scheme of a leaderless SMR protocol. First, the lower latency of the fast path in existing protocols comes with a downside: the fast path must involve a *fast quorum* of replicas bigger than a majority, which increases latency due to accesses to far-away replicas. For example, in Generalized Paxos [14] the fast quorum consists of at least $\frac{2r}{3}$ replicas, and in EPaxos of at least $\frac{3r}{4}$ replicas. To solve this problem, in `Atlas` the size of the fast quorum is a function of the number of allowed failures f – namely, $\lfloor \frac{r}{2} \rfloor + f$. Smaller values of f result in smaller fast quorums, thereby decreasing latency. `Atlas` thus trades off higher fault tolerance for lower latency. Note that violating the assumption the protocol makes about the number of failures may only compromise liveness, but never safety. In particular, if more than f transient outages occur, due to, e.g., connectivity problems, `Atlas` will just block until enough sites are reachable[1].

[1]Apart from data centers being down, geo-distributed systems may also exhibit network partitionings, which partition off

The second key change introduced by Atlas is that it can take the fast path even when conflicting commands are submitted concurrently, something that is not allowed by existing SMR protocols [11, 14]. This permits processing most commands via the fast path when the conflict rate is low-to-moderate, as is typical for SMR applications [3, 4]. Moreover, when $f = 1$, Atlas *always* takes the fast path and its fast quorum is a plain majority.

Simple recovery Failure recovery is the most subtle part of an SMR protocol with a fast path because the protocol needs to recover the decisions reached by the failed replicas while they were short-cutting some of the protocols steps in the fast path. This is only made more difficult with smaller fast quorums, as a failed process leaves information about its computations at fewer replicas. Atlas achieves its performant fast path while having a recovery protocol that is significantly simpler than that of previous leaderless protocols [10, 11]. We rigorously prove the correctness of this recovery mechanism.

Predictable performance Existing leaderless SMR protocols suffer from drawbacks in the way they order commands. Some protocols [9–11] (including Atlas), 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 [9, 19]. Other protocols [20, 21] 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.

Tempo 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*. Because it builds on Atlas, Tempo also offers the two features above, *low (average) latency* and *simple recovery* – Tempo is the first leaderless protocol to provide all three.

Tempo reuses most of Atlas techniques but, instead of maintaining explicit dependencies between commands, it assigns a scalar *timestamp* to each command and executes 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 [10, 21–23]. 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.

We organize this thesis as follows. Chapter 2 puts our work in context, covering the concept of SMR and relevant SMR protocols. Chapter 3 and Chapter 4 present the Atlas and Tempo protocols, respectively. Chapter 5 evaluates the protocols experimentally. Finally, Chapter 6 concludes this thesis.

several data centers from the rest of the system. Atlas may block for the duration of the partitioning, which is unavoidable due to the CAP theorem [18].

List of Publications

This thesis is based on the following papers published at EuroSys'20 [24] and at EuroSys'21 [25]:

Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, Pierre Sutra.
State-Machine Replication for Planet-Scale Systems.
European Conference on Computer Systems (EuroSys), Heraklion, Greece, 2020.

Vitor Enes, Carlos Baquero, Alexey Gotsman, Pierre Sutra.
Efficient Replication via Timestamp Stability.
European Conference on Computer Systems (EuroSys), Online, United Kingdom, 2021.

2

Background

In this chapter we put our work in context. We start by covering the system model in §2.1 and the concept of State-Machine Replication (SMR) in §2.2. This is followed by a detailed explanation of Paxos in §2.3, one of the most prominent SMR protocols. We then briefly cover other SMR protocols such as EPaxos and Caesar in §2.4 and Janus §2.5. Finally, we conclude the chapter with an overall comparison between `Atlas`, `Tempo` and the aforementioned protocols (§2.6), summarized in Table 2.1.

2.1 System Model

We consider an asynchronous distributed system where processes may fail by crashing but do not behave maliciously. The system consists of r processes $\mathbb{I} = \{1, \ldots, r\}$, of which at most f may fail. We assume that the set of processes is static. In this thesis we study how we can build a practical system that is geo-replicated among this set of processes. For that, we will leverage the concept of *consensus*, and so we start with a brief description of the consensus problem.

Consensus

Solving *consensus* entails having a set of processes reaching agreement on some value. Consensus is one of the most fundamental problems in distributed systems and has many applications, e.g., deciding whether to commit or abort a transaction, or decide which command to pick for a certain entry of a replicated log (§2.2). A protocol that solves consensus ensures the following properties:

Validity. If a process decides on value v, then v was proposed by some process.

Agreement. If a process decides on value v and some other process decides on value v', then $v = v'$.

Liveness. All non-faulty process eventually decide on a value.

Having a protocol that ensures these three properties is simple if processes are reliable, but it is quite challenging in the presence of faults. Famously, the FLP theorem [26] states that no protocol can solve consensus in an asynchronous system prone to faults. Hence, to ensure Liveness we assume that the

network is eventually synchronous [27]. This model guarantees that the system becomes synchronous after some unknown time, with message delays between non-failed processes bounded by an unknown constant δ and with process clocks tracking real time. These timing assumptions allow us to implement Ω, the leader election failure detector [28], which ensures that from some point on, all correct processes nominate the same correct process as the leader. Ω will be leveraged by the Paxos liveness algorithm presented later in this chapter (§2.3) and by the `Tempo` liveness algorithm (§4.2.7).

2.2 State-Machine Replication

There is an inherent trade-off when building planet-scale replicated systems: if we cannot afford high latencies, then we have to settle for weak consistency; or similarly, if we have to provide strong consistency, then we have to give up low latencies. This trade-off has been famously stated by the CAP theorem [18] and some later variations such as the PACELC theorem [29] and the CAP critique [30] that better capture the latency aspect of the trade-off. Systems providing strong consistency prevent anomalies (e.g., unexpected behavior visible to users) and reduce programming complexity [4, 31, 32]. In this thesis, we focus on these.

The strongest consistency level that replicated systems can offer is *linearizability* [6]. Informally, a system is linearizable if it appears as if commands submitted by clients are executed by a sequential process in an order consistent with the *real-time order*, i.e., the order of non-overlapping command invocations. For example, if some command x by client A returns before some command y by client B is submitted, then x must appear before y in the linearization order. However, if y is submitted in between x being submitted and returned, then commands x and y can be ordered in any way (since their invocations overlap).

State-Machine Replication (SMR) is a common way of implementing linearizable replicated services [5, 8, 22]. A service is defined by a deterministic state machine accepting a set of commands. Each process maintains a local copy of the state machine and applies commands in an order determined by an SMR protocol. In its most basic form, an SMR protocol decides on a sequence of commands. This sequence is also known as the *log*. As processes start from the same initial state and execute commands in the order dictated by log, this simple mechanism ensures that processes stay in sync.

2.3 Paxos

Paxos [7, 33] is one of the most used SMR protocols. Detailed descriptions of the protocol can be found in many recent works [34–38]. In order to aid understanding `Atlas` (§3) and `Tempo` (§4), which use Paxos as a building block, we also give a review of the protocol.

For each entry of the log (§2.2), Paxos ensures that a single command is *decided* for it, i.e., that all processes agree on which command such entry should have. For this, Paxos uses a single-decree protocol

that we present next. This protocol solves the consensus problem (§2.1). In Paxos the system consists of r processes $\mathbb{I} = \{1, \dots, r\}$, of which at most $f = \lfloor \frac{r-1}{2} \rfloor$ may fail.

Consensus protocol

Algorithm 2.1 specifies the single-decree Paxos protocol at a process $i \in \mathbb{I}$. In Paxos there is an unbounded collection of ballot numbers. Ballot numbers are totally ordered and are allocated to processes round-robin. If for example we have 3 processes, we allocate ballots $\{1, 4, 7, \dots\}$ to process 1, ballots $\{2, 5, 8, \dots\}$ to process 2 and ballots $\{3, 6, 9, \dots\}$ to process 3. Paxos ensures that a single command is decided by maintaining the following invariant:

Invariant 2.1. Assume that command c is decided at some ballot b. If command c' is decided at some ballot $b' > b$, then $c' = c$.

Paxos maintains this invariant with a two-phase protocol. In Paxos phase 1, a process leading with some ballot b *(i)* finds out which commands could have been decided at a ballot lower than b (if any), and *(ii)* prevents any command to be decided at a ballot lower than b. In Paxos phase 2, the process first computes its proposal for the command: either a command that could have been decided at a lower ballot (discovered during phase 1), or, if no command could have been decided, any command. The process then sends this proposal to all processes, and if enough processes accept it, the command is considered as decided. To better match the specification of `Atlas` and `Tempo`, Algorithm 2.1 denotes Paxos phase 1 messages by `MRec` and Paxos phase 2 messages by `MConsensus`[1].

Every process stores the ballot number `bal` it is currently participating in and the last ballot `abal` in which it accepted a consensus proposal (if any). Initially `bal` $=$ `abal` $= 0$. In order to get a command decided, a process first calls the function **recover** (line 1). In this function, the process picks a ballot number it owns higher than any it participated so far (line 2) and starts Paxos phase 1 by sending an `MRec` message with this ballot to all processes (line 3).

Upon the receipt of such a message, a process accepts the `MRec` message only if the ballot in the message is greater than its `bal` (line 5). The process sets its `bal` to the received ballot. This ensures that, during phase 2, the process will not accept a consensus proposal at a ballot lower than the ballot received. Then, it replies with an `MRecAck` message containing the command (`cmd`) and the ballot at which it was previously accepted (`abal`). Note that `abal` $= 0$ if the process has not yet accepted any consensus proposal for the command.

Once a process receives `MRecAck` messages from a *majority quorum* (i.e., $\lfloor \frac{r}{2} \rfloor + 1$ processes) (line 8), it starts Paxos phase 2 by sending an `MConsensus` message containing its proposal for the command. To ensure Invariant 2.1, this proposal may have to be an earlier proposal by another process. Thus, the process first inspects whether a consensus proposal could have been decided by checking if any of the processes replied with a non-zero `abal` (line 10). If so, the process selects the consensus proposal

[1] `MRec` is commonly known by Paxos message 1a, `MRecAck` by 1b, `MConsensus` by 2a and `MConsensusAck` by 2b.

Algorithm 2.1: Single-decree Paxos protocol at process $i \in \mathbb{I}$.

```
1  recover()
2      b ← i + r(⌊(bal−1)/r⌋ + 1)
3      send MRec(b) to 𝕀
4  receive MRec(b) from j
5      pre: bal < b
6      bal ← b
7      send MRecAck(abal, cmd, b) to j
8  receive MRecAck(ab_j, c_j, b) from ∀j ∈ Q
9      pre: bal = b ∧ |Q| = ⌊r/2⌋ + 1
10     if ∃k ∈ Q. ab_k ≠ 0 then
11         let k be such that ab_k is maximal
12         send MConsensus(c_k, b) to 𝕀
13     else
14         send MConsensus(some_cmd(), b) to 𝕀
15 receive MConsensus(c, b) from j
16     pre: bal ≤ b
17     cmd ← c; bal ← b; abal ← b
18     send MConsensusAck(b) to j
19 receive MConsensusAck(b) from Q
20     pre: bal = b ∧ |Q| = ⌊r/2⌋ + 1
21     decide(cmd)
```

accepted at the highest ballot (line 11). If no consensus proposal could have been accepted before (i.e., if all processes replied with a zero `abal`), then the process is free to propose any command (e.g., a command submitted by a client) (line 14).

A process accepts an `MConsensus` message only if its `bal` is not greater than the ballot in the message (line 16). Then it stores the proposal in `cmd`, sets `bal` and `abal` to the ballot in the message, and replies with an `MConsensusAck` message. Note that $\texttt{abal} \neq 0$ after a process accepts a consensus proposal. As mentioned above, this fact is used by processes executing the `MRecAck` handler.

Once a process gathers `MConsensusAck` messages by a majority quorum (line 20), it can finally decide this proposal for the command (line 21). This is safe because any majority quorum used for Paxos phase 1 must intersect with any majority quorum used for Paxos phase 2. Thus, after Paxos phase 2, the process is sure that any other process trying to decide a command will find out about this proposal during Paxos phase 1.

Skipping Paxos phase 1

By design, line 2 only computes ballots greater than r. This allows us to have one special ballot $s < r$ that can skip Paxos phase 1. During Paxos phase 1 with some ballot b, a process tries to find out about proposals smaller than b that could have been accepted. Because $s < r$ and ballots computed in line 2 are greater than r, no proposal smaller s can ever be accepted, and thus this step is unnecessary for

ballot s. Moreover, when accepting a Paxos phase 1 message with some ballot b, a process promises to not accept a consensus proposal with a lower ballot. Again, this is not necessary for ballot s as ballots from line 2 are greater than r. For these two reasons, process s leading with ballot s can safely skip Paxos phase 1. In this case, process s does not have to call **recover** in order to later propose a command – it can simply send an `MConsensus` message containing its proposal. This mechanism is leveraged by `Atlas` (§3) and `Tempo` (§4) in their slow paths.

Multi-Paxos optimization

In Paxos, a command is decided after 4 message delays – 2 message delays for each phase of Paxos. However, practical implementations of Paxos employ the Multi-Paxos [7, 33] optimization which allows commands to be decided after only 2 message delays. First, note that Paxos phase 1 can be performed for any entry of the log independently of the command to be proposed for such entry. Thus, in Multi-Paxos, a process performs phase 1 for all entries of the log. If it succeeds, this process is now called the *leader* – this mechanism is called *leader election*. When the elected leader receives a command issued by a client, it first selects the next available log entry, and then simply performs Paxos phase 2 by sending an `MConsensus` message containing the command received. Once it gathers enough `MConsensusAck` messages, it decides the command as usual, effectively reducing the number of message delays from 4 to 2.

Flexible Paxos optimization

As observed in Flexible Paxos (FPaxos) [35], Paxos safety relies on the fact that any phase 1 quorum intersects with any phase 2 quorum. Paxos ensures this intersection requirement with majority quorums for both phases, but the requirement can be met with other quorum sizes. For example, if we have $r - f$ processes for phase 1 and $f + 1$ processes for phase 2, then any phase 1 quorum must also intersect with any phase 2 quorum. In this scheme, if $f = \lfloor \frac{r-1}{2} \rfloor$ (as assumed by Paxos), we have again majority quorums for both phases (when r is odd). However, if we allow f to be any value such that $1 \leq f \leq \lfloor \frac{r-1}{2} \rfloor$, we can have one of the Paxos phases with a much smaller quorum. For example, with $r = 7$ and $f = 1$, FPaxos phase 1 quorums would be of size $r - f = 6$ and phase 2 quorums of size $f + 1 = 2$.

The above observation in FPaxos allows the leader elected in phase 1 to contact fewer processes during phase 2. In planet-scale systems these processes might be located far away from the leader, and thus this strategy can reduce the time the leader takes to complete phase 2. For this reason, the overall protocol latency may be lowered since, in Multi-Paxos, Paxos phase 2 is much more frequent than Paxos phase 1. In `Atlas` (§3) and `Tempo` (§4) we leverage this observation to reduce the quorum size of slow paths at the expense of larger quorums during recovery.

Algorithm 2.2: Paxos liveness protocol at process $i \in \mathbb{I}$.

```
22 periodically
23   if leader = i ∧ (bal = 0 ∨ bal_leader(bal) ≠ i) then
24     recover()
25 receive MConsensus(_, b) or MRec(b) from j
26   pre: bal > b
27   send MRecNAck(bal) to j
28 receive MRecNAck(b)
29   pre: leader = i ∧ bal < b
30   bal ← b
31   recover()
32 bal_leader(b)
33   return b − r ∗ ⌊(b−1)/r⌋
```

Liveness protocol

The Paxos rules we have covered so far ensure that the protocol is always safe, independently of how the whole system behaves (e.g., which processes call function **recover**, whether messages are lost, duplicated, reordered, and so on). However, if processes are allowed to call **recover** in an unconstrained way, the protocol is not live, i.e., it may happen that no command is ever decided (even in a synchronous network). Algorithm 2.2 addresses this issue and specifies a liveness protocol for Paxos. Again, we give a detailed description of this mechanism as it is used as a building block by Tempo in its liveness protocol (§4.2.7). This mechanism uses Ω, the leader election failure detector [28], which ensures that from some point on, all correct processes nominate the same correct process as the leader (§2.1). In Algorithm 2.2, the variable `leader` denotes the current leader nominated at process i. We say that `leader` stabilizes when it stops changing at all correct processes.

Algorithm 2.2 refines Algorithm 2.1 by specifying a particular policy for invoking **recover** at a process $i \in \mathbb{I}$. Process i is allowed to invoke **recover** at line 23 only if it is the leader according to `leader`. Furthermore, it only invokes **recover** at line 23 either if it has not yet participated in consensus (i.e., `bal` $= 0$) or, if it did, the consensus was lead by another process (i.e., $\textbf{bal\_leader}(\texttt{bal}) \neq i$). In particular, process i does not invoke **recover** at line 23 if it is the leader of `bal` (i.e., if $\textbf{bal\_leader}(\texttt{bal}) = i$). This ensures that process i does disrupt a recovery lead by itself.

For a leader to make progress with some $\texttt{MRec}(b)$ message, it is required that $\lfloor \frac{r}{2} \rfloor + 1$ processes (line 8) have their `bal` $< b$ (line 5). This may not always be the case because, before the variable `leader` stabilizes, any process can invoke **recover** at line 23 if it thinks it is the leader. To help the leader select a high enough ballot, and thus ensure it will make progress, we introduce a new message type, `MRecNAck`. A process sends an $\texttt{MRecNAck}(\texttt{bal})$ at line 27 when it receives an $\texttt{MConsensus}(\_, b)$ or $\texttt{MRec}(b)$ (line 25) with some ballot b lower than its `bal` (line 26). When process i receives an $\texttt{MRecNAck}(b)$ with some ballot number b higher than its `bal`, if it is still the leader (line 29), it joins ballot b (line 30) and

invokes **recover** again. This results in process i sending a new `MRec` with some ballot higher than the received b but lead by itself. As only `leader` is allowed to invoke **recover** at line 23 and line 29, and since `leader` eventually stabilizes, this mechanism ensures that eventually the stable leader will start a high enough ballot in which enough processes will participate.

2.4 EPaxos and Caesar

In the previous section we have covered Paxos, one of the most prominent leader-based protocols. With its Multi-Paxos optimization, after a leader is elected, commands can be decided after 2 message delays. However, this assumes that the client is co-located with the elected leader. When the client is faraway from the Paxos leader, additional message delays are required.

Now we shift our focus to leaderless protocols, briefly covering EPaxos [11] and Caesar [10]. Because these protocols do not have a distinguished leader, clients can submit their commands directly to the closest process in the system (not some leader process that might be faraway), potentially reducing the overall latency. This closest process is called a *coordinator*. Leaderless protocols offer a *fast path* where commands are decided after 2 message delays, and fallback to a *slow path* when the fast path is unsafe, deciding commands after a total of 4 message delays.

Leaderless protocols typically exploit the fact that commands in SMR applications frequently commute [3, 4], and for the replicated state machine to be linearizable, it is enough that replicas only agree on the order of non-commuting commands [13, 14]. This permits a command to take the fast path e.g. when the command commutes with all commands concurrently submitted to the system. In the presence of concurrent non-commuting (aka conflicting) commands, usually the slow path has to be taken.

EPaxos

EPaxos [11] ensures that processes agree on the order of conflicting commands by associating each command with a set of dependencies. We say that a command is *committed* when processes agree on its dependencies. The sets of committed dependencies are then used to build a directed graph. Due to the way that dependencies are computed in EPaxos (see below), this dependency graph may be cyclic, and thus 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. Cycles in the components are broken in some deterministic way (e.g., order by command identifier). We cover this execution mechanism in more detail in §3.2.3 when presenting `Atlas`. The size of the strongly connected components is a priori unbounded. In fact, as we show in §4.2.4, there are pathological scenarios where the protocol continuously commits commands but can never execute them, even under a synchronous network [11, 19]. For this reason, EPaxos does not ensure liveness. `Atlas` leverages the same execution mechanism of EPaxos, and thus, it does not ensure liveness either. This is not only a theoretical issue, as in practice these protocols offer high tail latencies (§5).

As mentioned above, EPaxos *commits* each command c with a set of dependencies $\texttt{dep}[c]$. For this, when the coordinator receives command c, it first forwards it to a *fast quorum* of size $\lfloor \frac{3r}{4} \rfloor$. Then, when a fast-quorum process receives command c, it computes a set of conflicts, i.e., the set of commands previously processed that do not commute with c. These sets of conflicts are then sent to the coordinator. Once the coordinator receives the conflicts by all fast-quorum processes, if all sets of conflicts are equal, then the fast path is taken. If not, the coordinator is forced to take a slow path, contacting a *slow quorum* of size $\lfloor \frac{r}{2} \rfloor + 1$. In both cases, the command is committed with $\texttt{dep}[c]$ equal to the union of all sets of conflicts reported by the fast quorum. Note that the slow path mechanism performs the equivalent of Paxos phase 2. As Paxos phase 2 quorums must intersect with phase 1 quorums during recovery (in case of coordinator failure), EPaxos employs a *recovery quorum* also of size $\lfloor \frac{r}{2} \rfloor + 1$. Just like Paxos, EPaxos assumes that the number of allowed failures (f) is always $\lfloor \frac{r-1}{2} \rfloor$, and thus f is not configurable as in Flexible Paxos.

Caesar

Caesar [10] commits each command c not only with a set of dependencies $\texttt{dep}[c]$, but also with a unique timestamp $\texttt{ts}[c]$. Commands are executed in timestamp order, and dependencies are used to determine the predecessors of a command. Timestamps and dependencies are combined in Caesar with the goal of improving performance in workloads with a low-to-moderate conflict rate. The quorum sizes are similar to EPaxos with the exception of the size of fast quorums ($\lceil \frac{3r}{4} \rceil$). We cover the protocol in more detail in §4.2.3. We also show in §4.2.4 that Caesar allows pathological scenarios where commands are never committed at all, and thus Caesar does not ensure liveness either.

2.5 Janus

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 r processes in a given partition. When processing commands accessing a single partition, Janus behaves just like EPaxos. When processing a multi-partition command c, Janus commits the command in each partition individually, and then sets $\texttt{dep}[c]$ to the union of the dependencies committed per partition[2]. Since Janus is based on EPaxos, it also does not ensure liveness (§2.4).

Once $\texttt{dep}[c]$ is known, a process tries to build a strongly connected component containing c, just like in EPaxos. However, a process may not have all the dependencies it needs, and may have to find them out from other processes that do not replicate c (see the example below). For this reason, Janus is not *genuine*. We say that a protocol is genuine when, for every command c, only the processes that replicate c take steps to order and execute it [39].

[2] Janus was not originally presented exactly like this. However, the dependencies computed are the same when one sees the protocol this way. Our implementation of Janus is based on `Atlas`, not on EPaxos, and takes this interpretation (§5).

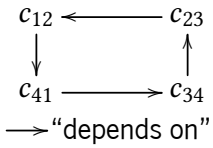


Figure 2.1: Dependency graph containing a cycle between 4 multi-partition commands: c_{12}, c_{23}, c_{34} and c_{41}.

To better understand why Janus is not genuine, consider the following example. Let us consider four partitions: 1, 2, 3 and 4. Consider also four multi-partition commands: c_{12} that accesses partitions 1 and 2, c_{23} that accesses partitions 2 and 3, c_{34} that accesses partitions 3 and 4, and c_{41} that accesses partitions 4 and 1. Assume that partition 1 receives first c_{41} and then c_{12}, partition 2 receives first c_{12} and then c_{23}, partition 3 receives first c_{23} and then c_{34}, and partition 4 receives first c_{34} and then c_{41}. This arrival order results in commands being committed with the following dependencies: $\mathtt{dep}[c_{12}] = \{c_{41}\}$, $\mathtt{dep}[c_{23}] = \{c_{12}\}$, $\mathtt{dep}[c_{34}] = \{c_{23}\}$, and $\mathtt{dep}[c_{41}] = \{c_{34}\}$. These dependencies form the strongly connected component depicted in Figure 2.1.

Let p be some process from partition 1 that wants to order and execute c_{41}. For that, p has to build the strongly connected component in Figure 2.1. Like in EPaxos (§2.4), once the component is built, cycles should be broken in some deterministic way (e.g., order by command identifier). Finally, following this deterministic order, process p has to execute only the commands from the component that are replicated by itself, i.e., c_{41} and c_{12}. Note that process p does not replicate c_{34}, the dependency of c_{41}, and thus, it must inquire a process that does. When p receives such information, it learns that $\mathtt{dep}[c_{34}] = \{c_{23}\}$. Now p knows about c_{34}, but because it does not replicate its dependency c_{23}, it must perform another inquiry. Note that process p initially wanted to order and execute c_{41}, which is replicated by partitions 4 and 1. For the protocol to be genuine, process p could only contact these two partitions to order and execute c_{41}. However, p needs information about c_{23}, which is replicated only by partitions 2 and 3, and these partitions are not accessed by c_{41}.

Note that these inquires by p are necessary for a correct ordering of the commands. Given only the information p had before the inquires, i.e., $\mathtt{dep}[c_{12}] = \{c_{41}\}$ and $\mathtt{dep}[c_{41}] = \{c_{34}\}$, p would execute first c_{41} and then c_{12} (as the latter depends on the former). However, this may not be the correct execution order. To understand why, assume that cycles are broken by ordering commands using their identifiers, and that the command identifiers are ordered in the following way: $c_{12} < c_{23} < c_{34} < c_{41}$. In this case, the correct execution order would instead be c_{12} and then c_{41}. This shows why these inquires are necessary for a correct command ordering.

	quorum sizes			configurable f	live	genuine
	fast	slow (phase 2)	recovery (phase 1)			
Flexible Paxos	–	$f+1$	$r-f$	✓	✓	–
EPaxos	$\lfloor \frac{3r}{4} \rfloor$	$\lfloor \frac{r}{2} \rfloor + 1$	$\lfloor \frac{r}{2} \rfloor + 1$	✗	✗	–
Caesar	$\lceil \frac{3r}{4} \rceil$	$\lfloor \frac{r}{2} \rfloor + 1$	$\lfloor \frac{r}{2} \rfloor + 1$	✗	✗	–
Janus	r	$\lfloor \frac{r}{2} \rfloor + 1$	$\lfloor \frac{r}{2} \rfloor + 1$	✗	✗	✗
`Atlas` (§3)	$\lfloor \frac{r}{2} \rfloor + f$	$f+1$	$r-f$	✓	✗	–
`Tempo` (§4)	$\lfloor \frac{r}{2} \rfloor + f$	$f+1$	$r-f$	✓	✓	✓

Table 2.1: Comparison between SMR protocols.

2.6 Protocol Comparison Summary

In Table 2.1 the compare `Atlas` and `Tempo` with each protocol covered so far. We analyze several dimensions. First we consider the size of fast, slow and recovery quorums. Flexible Paxos does not have a fast quorum size as the protocol does not employ a fast path. `Atlas` and `Tempo` have fast quorums of size $\lfloor \frac{r}{2} \rfloor + f$ so that smaller values of f result in smaller fast quorums.

We also analyze whether protocols allow the number of failures (f) to be configurable – `Atlas` and `Tempo` are the only leaderless protocols that do. As for liveness, `Atlas` inherits the liveness issue of EPaxos that we describe in §4.2.4, and thus `Atlas` is marked as not live. `Tempo` builds on `Atlas` but is able to solve this liveness issue, being the only leaderless protocol in Table 2.1 that ensures liveness. Finally, for the two protocols designed to support partial replication, Janus and `Tempo`, we also mark them also genuine or not.

3

ATLAS: Low-Latency, Simple Leaderless Consensus

In this chapter we present Atlas, a leaderless SMR protocol specially tailored to planet-scale systems that provides *low average latency* and employs a *simple recovery* mechanism. Like previous protocols such as EPaxos and Caesar (§2), Atlas is *leaderless*, i.e., it orders commands in a decentralized way, without relying on a distinguished leader site. This improves availability and allows serving clients with the same quality of service independently of their geographical locations. As is common, Atlas also exploits the fact that commands in SMR applications frequently commute [3, 4], and for the replicated state machine to be linearizable, it is enough that replicas only agree on the order of non-commuting commands (§2.2). This permits processing a command in one round trip from the closest replica using a *fast path*, e.g., when the command commutes with all commands concurrently submitted for execution. In the presence of concurrent non-commuting commands, the protocol may sometimes have to take a *slow path*, which requires two round trips.

Low-latency Atlas employs two techniques in order to provide low average latency. First, and following the ideas of Flexible Paxos (§2.3), Atlas allows choosing the maximum number of sites that can fail (f) independently of the overall number of sites (r). This is motivated by the fact that common SMR protocols allow any minority of sites to fail simultaneously: e.g., deploying a typical protocol with $r = 11$ sites allows the concurrent failure of $f = 5$ of them. However, natural disasters leading to the loss of a data center are infrequent, and temporary data center outages rarely happen concurrently (§5). Industry practitioners thus assume small values for f in geo-distributed systems, e.g., 1 or 2 [4]. Atlas exploits this fact to offer low latency. The size of the fast quorum in Atlas is a function of the number of allowed failures f – namely, $\lfloor \frac{r}{2} \rfloor + f$ – and thus, smaller values of f result in smaller fast quorums, thereby decreasing latency. This is not allowed by prior protocols with a fast path, which have fast quorum sizes that are simply a function of the number of sites r. For example, the fast quorum consists of at least $\frac{2r}{3}$ sites in Generalized Paxos [14] and at least $\frac{3r}{4}$ sites in EPaxos (§2).

Secondly, Atlas can take the fast path even when non-commuting commands are submitted concurrently, something that is not allowed by Generalized Paxos and EPaxos. This flexibility permits processing most commands via the fast path when the conflict rate is low-to-moderate, as is typical for SMR applications [3, 4].

Simple recovery The biggest challenge we faced in achieving the above features – smaller fast quorums and a flexible fast-path condition – was in designing a correct failure recovery mechanism. Failure recovery is the most subtle part of an SMR protocol with a fast path because the protocol needs to recover the decisions reached by the failed replicas while they were short-cutting some of the protocols steps in the fast path. This is only made more difficult with smaller fast quorums, as a failed process leaves information about its computations at fewer replicas. Atlas achieves its performant fast path while having a recovery protocol that is significantly simpler than that of previous leaderless protocols [10, 11]. We rigorously prove the correctness of this recovery mechanism.

We organize this chapter as follows. We start by covering the system model in §3.1. This is followed by a detailed explanation of the Atlas protocol in §3.2: we cover the commit protocol in §3.2.1, the recovery protocol in §3.2.2, the execution protocol in §3.2.3, a comparison with EPaxos in §3.2.4, and a few optimizations in §3.2.5. In §3.3 we show the correctness of Atlas. We conclude this chapter with a summary and related work in §3.4.

3.1 System Model

We consider an asynchronous distributed system where processes may fail by crashing but do not behave maliciously. The system consists of r processes $\mathbb{I} = \{1, \ldots, r\}$, of which at most f may fail. Following Flexible Paxos [35], f can be any value such that $1 \leq f \leq \lfloor \frac{r-1}{2} \rfloor$. This allows using small values of f regardless of the replication factor r, which is appropriate in geo-replication [4, 17]. In a geo-distributed deployment, each process represents a data center, so that a failure corresponds to the outage of a whole data center. Failures of single machines are orthogonal to our concerns and can be masked by replicating a process within a data center using standard techniques [7, 8]. We assume that the set of processes is static. Classical approaches can be used to add reconfiguration to Atlas [7, 11]. Reconfiguration can also be used in practice to allow processes that crash and recover to rejoin the system.

State-Machine Replication (SMR) is a common way of implementing strongly consistent replicated services (§2). A service is defined by a deterministic state machine accepting a set of *commands*, denoted by 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. The protocol provides a command $\texttt{submit}(c)$, which allows a process to submit a command $c \in C$ for execution on behalf of a client. The protocol may also trigger an event $\texttt{execute}(c)$ at a process, asking it to apply c to the local service replica; after execution, the process that submitted

the command may return the outcome of c to the client. Without loss of generality, we assume that each submitted command is unique.

The strongest property a replicated service implemented using SMR may satisfy is *linearizability* (§2). As observed in [13, 14], for the replicated service to be linearizable, the SMR protocol does not need to ensure that commands are executed at processes in the exact same order: it is enough to agree on the order of non-commuting commands.

We now give the specification of the SMR protocol. We say that commands c and d *commute* if in every state s of the state machine: *(i)* executing c followed by d or d followed by c in s leads to the same state; and *(ii)* c returns the same response in s as in the state obtained by executing d in s, and vice versa. If commands c and d do not commute, we say that they *conflict*[1] and denote it by $\texttt{conflict}(c, d)$. Given two commands c and d, we write $c \mapsto_i d$ when $\texttt{conflict}(c, d)$ and c is executed before d at some process $i \in \mathbb{I}$. We also define the following *real-time order*: $c \rightsquigarrow d$ if the command c returns before the command d was submitted. Let $\mapsto = (\bigcup_{i \in \mathbb{I}} \mapsto_i) \cup \rightsquigarrow$. An SMR 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 $\mapsto$ is acyclic.

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

3.2 The Atlas Protocol

To aid understanding, we first illustrate by example the message flow of the Atlas protocol, which corresponds to a common structure of leaderless SMR protocols [11]. We then describe the protocol in detail (§3.2.1-§3.2.5).

Figure 3.1 illustrates how Atlas processes two conflicting commands, $\mathbf{a}$ and $\mathbf{b}$, with $r = 5$ processes and at most $f = 2$ failures. At a given process, a command usually goes through several *phases*: the initial phase START, then COLLECT, COMMIT and EXECUTE (an additional phase RECOVER is used when handling failures). We summarize these phases and allowed phase transitions in Figure 3.2.

Command $\mathbf{a}$ starts its journey when $\texttt{submit}(\mathbf{a})$ is invoked at process 1. We call process 1 the initial *coordinator* of $\mathbf{a}$. This coordinator is initial because, if it fails or is slow, another process may take over. Command $\mathbf{a}$ then enters the COLLECT phase at process 1, whose goal is to compute the set of commands

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

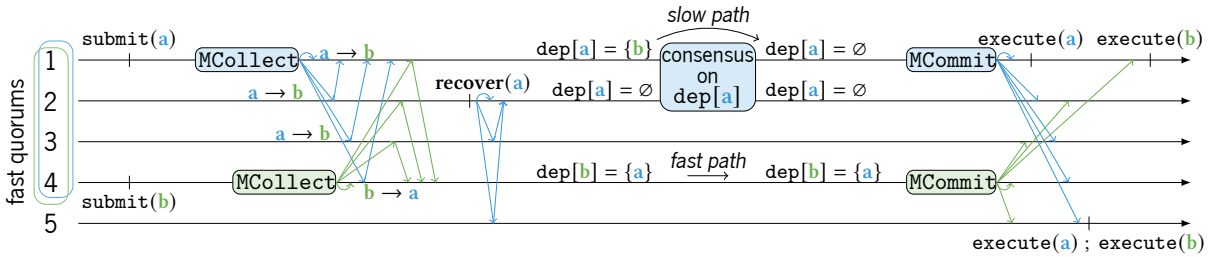


Figure 3.1: Example of processing two conflicting commands a and b in Atlas with $r = 5$ processes and up to $f = 2$ failures. We omit the messages implementing consensus and depict this step abstractly by the consensus box.

that are *dependencies* of a, denoted by dep[a]. These dependencies are later used to determine the order of execution of conflicting commands. To compute dependencies, process 1 sends an MCollect message containing command a to a *fast quorum* of processes, which is at least a majority but may be bigger. In our example the fast quorum picked by 1 is $\{1, 2, 3, 4\}$.

Each process in the fast quorum returns the set of commands conflicting with a that it received before a. In Figure 3.1, $\longrightarrow$ indicates the order in which processes receive commands. For instance, process 4 receives b first, whereas 1, 2 and 3 do not receive any command before a. Based on the replies, process 1 computes the value of dep[a] as $\{b\}$, the union of all conflicting commands reported.

If a coordinator of a command is suspected to have failed, another process may try to take over. In Figure 3.1, process 2 suspects 1 and becomes another coordinator of a, denoted by **recover**(a). Process 2 contacts a majority quorum of processes $\{2, 3, 5\}$ and computes its own version of the dependencies of a: dep[a] $= \emptyset$.

Dependencies are used to determine the order in which conflicting commands are executed, and all processes have to execute conflicting commands in the same order. To ensure this, the coordinators of command a need to reach a consensus on the value of dep[a]. This is implemented using an optimized variant of single-decree Paxos (§2.3), with all r processes acting as acceptors. In our example, this makes the processes agree on dep[a] $= \emptyset$. The use of consensus represents the *slow path* of the protocol.

If a coordinator can ensure that all the values that can possibly be proposed to consensus are the same, then it can take the *fast path* of the protocol, avoiding the use of consensus. In Figure 3.1, this is the case for process 4 coordinating command b. For a process to take the fast path, we require it to receive a response from every process in the fast quorum, motivating the name of the latter.

After consensus or the shortcut via the fast path, a coordinator of a command sends its final dependencies to other processes in an MCommit message. A process stores these dependencies and marks the command as having entered the COMMIT phase. A command can be executed (and thereby transition to the EXECUTE phase) only after all its dependencies are in the COMMIT or EXECUTE phases. Since in our example dep[a] $= \emptyset$, processes can execute command a right after receiving its final dependencies ($\emptyset$). This is exploited by processes 1 and 2 in Figure 3.1. However, as dep[b] $= \{a\}$, processes must

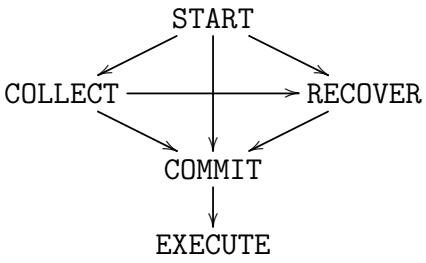


Figure 3.2: Command journey through phases in Atlas.

delay the execution of b until a is executed. This is the case for processes 3, 4 and 5 in Figure 3.1. Such an execution mechanism guarantees that the conflicting commands a and b are executed in the same order at all processes.

3.2.1 Commit Protocol

Algorithm 3.1 specifies the Atlas commit protocol at process $i \in \mathbb{I}$ in the failure-free case. We assume that self-addressed protocol messages are delivered immediately.

Start phase

A client submits a command $c \in C$ by invoking $\texttt{submit}(c)$ at one of the processes running Atlas, which will serve as the initial command coordinator. When $\texttt{submit}(c)$ is invoked at a process i (line 1), this coordinator first assigns to command c a unique identifier using a function next_id. In the following we denote the set of all identifiers by $\mathcal{D}$. We summarize in Table 3.1 the data maintained by each process for a command with identifier $id \in \mathcal{D}$. In particular, the mapping cmd stores the payload of the command, and the mapping phase tracks the progress of the command through phases. For brevity, the name of the phase written in lower case also denotes all the identifiers in that phase, e.g., $start = \{id \in \mathcal{D} \mid \texttt{phase}[id] = \texttt{START}\}$.

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

Invariant 3.1. For any two messages $\texttt{MCommit}(id, c, D)$ and $\texttt{MCommit}(id, c', D')$ sent, $c = c'$ and $D = D'$.

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

Algorithm 3.1: Atlas commit protocol at process $i \in \mathbb{I}$.

```
1  submit(c)
2      id ← next_id()
3      past ← conflicts(c)
4      Q ← fast_quorum(i)
5      send MCollect(id, c, past, Q) to Q
6  receive MCollect(id, c, past, Q) from j
7      pre: id ∈ start
8      dep[id] ← conflicts(c) ∪ past
9      cmd[id] ← c; quorum[id] ← Q
10     phase[id] ← COLLECT
11     send MCollectAck(id, dep[id]) to j
12 receive MCollectAck(id, dep_j) from ∀j ∈ Q
13     pre: id ∈ collect ∧ Q = quorum[id]
14     D ← ⋃_Q dep
15     if ⋃_Q dep = ⋂^f_Q dep then
16         send MCommit(id, cmd[id], D) to 𝕀
17     else
18         send MConsensus(id, cmd[id], D, i) to 𝕀
19 receive MConsensus(id, c, D, b) from j
20     pre: bal[id] ≤ b
21     cmd[id] ← c; dep[id] ← D
22     bal[id] ← b; abal[id] ← b
23     send MConsensusAck(id, b) to j
24 receive MConsensusAck(id, b) from Q
25     pre: bal[id] = b ∧ |Q| = f + 1
26     send MCommit(id, cmd[id], dep[id]) to 𝕀
27 receive MCommit(id, c, D)
28     pre: id ∉ commit ∪ execute
29     cmd[id] ← c; dep[id] ← D; phase[id] ← COMMIT
   ---------------------------------------------------
30 conflicts(c)
31     {id ∉ start | conflict(c, cmd[id])}
```

Invariant 3.2. Assume that messages $\texttt{MCommit}(id, c, D)$ and $\texttt{MCommit}(id', c', D')$ have been sent. If $id \neq id'$ and $\texttt{conflict}(c, c')$ then either $id' \in D$ or $id \in D'$, or both.

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

Collect phase

To compute the dependencies of a command c, its coordinator first computes the set of commands it knows about that conflict with c (denoted by *past*, line 3) using a function **conflicts**. The coordinator

Table 3.1: Atlas variables at a process.

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

then picks a fast quorum Q of size $\lfloor \frac{r}{2} \rfloor + f$ that includes itself (line 4) and sends an MCollect message with the information it computed to all processes in Q.

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

Once the coordinator receives an MCollectAck message from all processes in the fast quorum (line 13), it computes the dependencies for the command as the union of all reported dependencies $D = \bigcup_Q dep = \bigcup\{dep_j \mid j \in Q\}$ (line 14). Since a fast quorum contains at least a majority of processes, the following property implies that this computation maintains Invariant 3.2.

Property 3.1. Assume two conflicting commands with identifiers id and id' and dependencies D and D' computed as in line 14 over majority quorums. Then either $id' \in D$ or $id \in D'$, or both.

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

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

Slow path

If the coordinator of a command is suspected to have failed, another process may try to take over its job and compute a different set of dependencies. Hence, before an MCommit message is sent, processes must reach an agreement on its contents to satisfy Invariant 3.1. They can always achieve this by running

a consensus protocol – this is the slow path of Atlas. Consensus is implemented using single-decree (Flexible) Paxos [35]. 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 that try to take over. Every process stores for each identifier id the ballot number $\texttt{bal}[id]$ it is currently participating in and the last ballot $\texttt{abal}[id]$ in which it accepted a proposal (if any). Initially, $\texttt{bal}[id] = \texttt{abal}[id] = 0$.

When the initial coordinator i decides to go onto the slow path, it performs an analog of Paxos Phase 2: it sends an MConsensus message with its proposal and ballot i to a *slow quorum* that includes itself[2]. Following Flexible Paxos [35], the size of the slow quorum is only $f + 1$, rather than a majority like in classical Paxos. This minimizes the additional latency incurred on the slow path in exchange for using larger quorums in recovery (as described below). As usual in Paxos, a process accepts an MConsensus message only if its $\texttt{bal}[id]$ is not greater than the ballot in the message (line 20). Then it stores the proposal, sets $\texttt{bal}[id]$ and $\texttt{abal}[id]$ to the ballot in the message, and replies to the coordinator with MConsensusAck. Once the coordinator gathers $f + 1$ such replies (line 25), it is sure that its proposal will survive the allowed number of failures f, and it thus broadcasts the proposal in an MCommit message (line 26).

Fast path

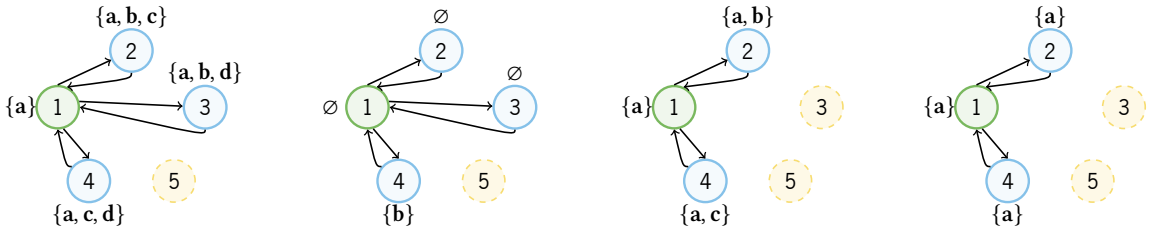
The initial coordinator of a command can avoid consensus when it can ensure that any process performing recovery will propose the same set of dependencies to consensus [40] – this is the fast path of Atlas, in which a command is committed after a single round trip to the closest fast quorum (line 16). In order to take the fast path, previous SMR protocols, such as Generalized Paxos [14] and EPaxos [11], require fast-quorum replies to match exactly. One of the key innovations of Atlas is that it is able to take the fast path even if this is not the case, e.g., when conflicting commands are submitted concurrently.

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

$$\bigcup\nolimits_Q^f dep = \{id \in \mathcal{D} \mid \texttt{count}(id) \geq f\};$$
$$\texttt{count}(id) = |\{j \in Q \mid id \in dep_j\}|.$$

Figure 3.3 contains several examples that illustrate the flexibility of the above fast-path condition. All examples consider $r = 5$ processes while tolerating varying numbers of faults f. The example in Figure 3.3a considers Atlas $f = 2$. The coordinator of some command, process 1, picks a fast quorum $Q = \{1, 2, 3, 4\}$ of size $\lfloor \frac{r}{2} \rfloor + f = 4$. It receives replies $dep_1 = \{\mathbf{a}\}$, $dep_2 = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$, $dep_3 = \{\mathbf{a}, \mathbf{b}, \mathbf{d}\}$, $dep_4 = \{\mathbf{a}, \mathbf{c}, \mathbf{d}\}$. The coordinator then computes $\bigcup_Q^2 dep = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$, i.e., all the dependencies reported at least twice. Since $\bigcup_Q dep = \bigcup_Q^2 dep$, the coordinator takes the fast path. This is not the

[2] As noted in §2.3, the initial coordinator i can safely skip Paxos Phase 1: since processes perform recovery with ballots higher than r, no proposal with a ballot lower than i can ever be accepted.



(a) ✓ Atlas $f = 2$ ✗ *matching replies* (b) ✗ Atlas $f = 2$ ✗ *matching replies* (c) ✓ Atlas $f = 1$ ✗ *matching replies* (d) ✓ Atlas $f = 1$ ✓ *matching replies*

Figure 3.3: Examples in which the fast path is taken ✓ or not ✗, for both Atlas and protocols that require *matching replies* from fast-quorum processes, such as EPaxos [11]. All examples consider $r = 5$ processes while tolerating f faults. The coordinator is always process 1, and circles with a solid line represent the processes that are part of the fast quorum. Next to each process we depict the set of dependencies sent to the coordinator (e.g., $\{\mathbf{a}, \mathbf{b}\}$).

case for the example in Figure 3.3b where $\bigcup_Q dep = \{\mathbf{b}\} \neq \emptyset = \bigcup^2_Q dep$ ($\mathbf{b}$ is excluded from $\bigcup^2_Q dep$ because $\mathtt{count}(\mathbf{b}) = 1$). In this case the coordinator has to take the slow path. Back in Figure 3.1 we had the same situation: coordinator 1 had to take the slow path because dependency $\mathbf{b}$ was declared solely by process 4. On the other hand, coordinator 4 was able to take the fast path, because dependency $\mathbf{a}$ was declared by $3 \geq f$ processes: 1, 2 and 3.

Notice that in Figure 3.3a, the coordinator takes the fast path even though dependencies reported by processes do not match, a situation which may arise when conflicting commands are submitted concurrently. Furthermore, when $f = 1$ we have $\{id \in \mathcal{D} \mid \mathtt{count}(id) < f\} = \emptyset$, so that the fast-path condition in line 15 always holds. Hence, Atlas $f = 1$ *always* takes the fast path, as is the case in Figures 3.3c and 3.3d. In contrast, EPaxos is able to take the fast path only in Figure 3.3d, since it is the only example in which fast-quorum replies match.

3.2.2 Recovery Protocol

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

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

Algorithm 3.2: Atlas recovery protocol at process $i \in \mathbb{I}$.

32 **recover**(id)
33 $\quad b \leftarrow i + r(\lfloor \frac{\texttt{bal}[id]-1}{r} \rfloor + 1)$
34 $\quad$ **send** MRec$(id, \texttt{cmd}[id], b)$ **to** $\mathbb{I}$
35 **receive** MRec$(id, _, _)$ **from** j
36 $\quad$ **pre:** $id \in commit \cup execute$
37 $\quad$ **send** MCommit$(id, \texttt{cmd}[id], \texttt{dep}[id])$ **to** j
38 **receive** MRec(id, c, b) **from** j
39 $\quad$ **pre:** $\texttt{bal}[id] < b \wedge id \notin commit \cup execute$
40 $\quad$ **if** $\texttt{bal}[id] = 0 \wedge id \in start$ **then**
41 $\quad\quad \texttt{dep}[id] \leftarrow \textbf{conflicts}(c)$; $\texttt{cmd}[id] \leftarrow c$
42 $\quad \texttt{bal}[id] \leftarrow b$
43 $\quad \texttt{phase}[id] \leftarrow$ RECOVER
44 $\quad$ **send** MRecAck$(id, \texttt{cmd}[id], \texttt{dep}[id], \texttt{quorum}[id], \texttt{abal}[id], b)$ **to** j
45 **receive** MRecAck$(id, c_j, dep_j, Q_j^0, ab_j, b)$ **from** $\forall j \in Q$
46 $\quad$ **pre:** $\texttt{bal}[id] = b \wedge |Q| = r - f$
47 $\quad$ **if** $\exists k \in Q.\, ab_k \neq 0$ **then**
48 $\quad\quad$ **let** k be such that ab_k is maximal
49 $\quad\quad$ **send** MConsensus(id, c_k, dep_k, b) **to** $\mathbb{I}$
50 $\quad$ **else if** $\exists k \in Q.\, Q_k^0 \neq \emptyset$ **then**
51 $\quad\quad I \leftarrow Q \cap Q_k^0$
52 $\quad\quad s \leftarrow \texttt{initial}_p(id) \in I$
53 $\quad\quad Q' \leftarrow$ **if** s **then** Q **else** I
54 $\quad\quad D \leftarrow \bigcup_{Q'} dep$
55 $\quad\quad$ **send** MConsensus(id, c_k, D, b) **to** $\mathbb{I}$
56 $\quad$ **else send** MConsensus$(id, \texttt{noOp}, \emptyset, b)$ **to** $\mathbb{I}$

the dependencies reported by the remaining $(\lfloor \frac{r}{2} \rfloor + f) - f = \lfloor \frac{r}{2} \rfloor$ processes. The following property captures this observation.

Property 3.2. Any fast-path proposal can be obtained by taking the union of the dependencies sent in MCollectAck by at least $\lfloor \frac{r}{2} \rfloor$ fast-quorum processes that are not the initial coordinator.

As an example, assume that after the fast path is taken in Figure 3.3a, $f = 2$ processes inside the fast quorum fail, one of them being the coordinator, process 1. Independently of which $\lfloor \frac{r}{2} \rfloor = 2$ fast-quorum processes survive, the proposal is always recovered by set union: $\bigcup_{\{2,3\}} dep = \bigcup_{\{2,4\}} dep = \bigcup_{\{3,4\}} dep = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}\}$.

In the case of Figure 3.3b it is unsafe to take the fast path since the proposal may not be recoverable: the failure of process 4 would lead to losing the dependency $\mathbf{b}$, since this dependency was reported exclusively by this process.

Recovery in detail

The recovery protocol of Atlas at a process $i \in \mathbb{I}$ is given in Algorithm 3.2. We use $\texttt{initial}(id)$ to denote a function that extracts from the command identifier id its initial coordinator.

A process takes over as the coordinator for some command with identifier id by calling $\textbf{recover}(id)$ (line 32 in Algorithm 3.2). In order to find out if a decision on the dependencies of id has been reached in consensus, the new coordinator first performs an analog of Paxos Phase 1 (§2.3). It picks a ballot number it owns higher than any it participated in so far (line 33) and sends an MRec message with this ballot to all processes.

Upon the receipt of such a message, in case id is already committed or executed (line 36), the process notifies the new coordinator with an MCommit message. Otherwise, as is standard in Paxos, the process accepts the MRec message only if the ballot in the message is greater than its $\texttt{bal}[id]$ (line 39). In this case, if the process is seeing id for the first time (line 40), it computes its contribution to id's dependencies as the set of conflicting commands (line 41). Then, the process sets $\texttt{bal}[id]$ to the new ballot and $\texttt{phase}[id]$ to RECOVER. Finally, the process replies with an MRecAck message containing all the information it has regarding id: the corresponding command (cmd), its current set of dependencies (dep), the ballot at which these were previously accepted (abal), and the fast quorum (quorum). Note that $\texttt{quorum}[id] = \emptyset$ if the process did not see the initial MCollect message, and $\texttt{abal}[id] = 0$ if the process has not yet accepted any consensus proposal.

In the MRecAck handler (line 45), the new coordinator computes its proposal given the information provided by processes and sends this proposal in an MConsensus message to all processes. As in Flexible Paxos, the new coordinator waits for $r - f$ MRecAck messages. This guarantees that, if a quorum of $f + 1$ processes accepted an MConsensus message with a proposal (which could have thus been sent in an MCommit message), the new coordinator will find out about this proposal. To maintain Invariant 3.1, if any process previously accepted a consensus proposal (line 47), by the standard Paxos rules [7, 35], the coordinator selects the proposal accepted at the highest ballot (line 48).

If no consensus proposal has been accepted before, the new coordinator checks whether any of the processes that replied has seen the initial MCollect message, by looking for any non-empty fast quorum (line 50). If the fast quorum is known, the new coordinator first computes at line 51 the set of processes I that belong both to the recovery quorum Q and the fast quorum Q_k^0. Then, depending on whether the initial coordinator replied or not, there are two possible cases that we describe next.

1) The initial coordinator replies to the new one ($s = true$, line 52). In this case the initial coordinator has not taken the fast path before receiving the MRec message from the new one, as it would have replied with MCommit instead of MRecAck (line 37). It will also not take the fast path in the future, since when processing the MRec message it sets the command phase to RECOVER (line 43), which invalidates the MCollectAck precondition (line 13). Since the initial coordinator never takes the fast path, the new coordinator can choose the command's dependencies in any way, as long as it maintains Invariant 3.2. By Property 3.1, this is satisfied if the coordinator chooses the set union of the dependencies declared by

at least a majority of processes. Hence, the new coordinator takes the union of the dependencies reported by the $r - f \geq r - \lfloor \frac{r-1}{2} \rfloor \geq \lfloor \frac{r}{2} \rfloor + 1$ processes in Q (line 54).

2) The initial coordinator does not reply to the new one ($s = false$, line 52). In this case the initial coordinator could have taken the fast path and, if it did, the new coordinator must propose the same dependencies. Given that the recovery quorum Q has size $r - f$ and the fast quorum Q_k^0 has size $\lfloor \frac{r}{2} \rfloor + f$, the set of processes $I = Q \cap Q_k^0$ (line 51) contains at least $\lfloor \frac{r}{2} \rfloor$ fast-quorum processes (distinct from the initial coordinator, as it did not reply). Furthermore, recall that when a process from I replies to the new coordinator, it sets the command phase to `RECOVER` (line 43), which invalidates the `MCollect` precondition (line 7). Hence, if the initial coordinator took the fast path, then each process in I must have processed its `MCollect` before the `MRec` of the new coordinator, and reported in the latter the dependencies from the former. Then using Property 3.2, the new coordinator recovers the fast-path proposal by taking the union of the dependencies from the processes in I (line 54). It can be shown that, even if the initial coordinator did not take the fast path, this computation maintains Invariant 3.2, despite I containing only $\lfloor \frac{r}{2} \rfloor$ processes and Property 3.1 requiring a majority of them. This is for the same reason this number of processes is sufficient in Property 3.2: dependencies declared by the initial coordinator are included into those declared by other fast-quorum processes (line 8).

It remains to address the case in which the process performing the recovery observes that no process saw the initial fast quorum, and consequently the submitted command (line 56). For instance, suppose that process i sends an $\texttt{MCollect}(id, c, \_, \_)$ only to process j and then fails. Further, assume that j receives another $\texttt{MCollect}(\_, c', \_, \_)$ from process k, replies with a dependency set that includes the identifier id of c, and also fails. Now, process k cannot execute c' without executing c (since c is a dependency of c'), and it cannot execute c because its payload has been lost. We solve this issue similarly to EPaxos: if a process takes over as the new coordinator and cannot find the associated payload, it may replace it by a special `noOp` command (line 56) that is not executed by the protocol and conflicts with all commands. With this, the final command for some identifier can take two possible values: the one submitted (line 1) or `noOp`. It is due to this that we include the command payload in addition to its dependencies into consensus messages associated with a given identifier (e.g., line 18), thus ensuring that a unique payload will be chosen (Invariant 3.1). Due to the possible replacement of a command by a `noOp`, the protocol actually ensures the following weakening of Invariant 3.2, which is still sufficient to establish its correctness.

Invariant 3.2′. Assume that messages $\texttt{MCommit}(id, c, D)$ and $\texttt{MCommit}(id', c', D')$ have been sent. If $id \neq id'$, $\texttt{conflict}(c, c')$, $c \neq \texttt{noOp}$ and $c' \neq \texttt{noOp}$, then either $id' \in D$ or $id \in D'$, or both.

3.2.3 Execution Protocol

Algorithm 3.3 describes a background task employed by `Atlas` that is responsible for executing commands after they are committed. This task runs in an infinite loop trying to execute a *batch* of commands. We define a batch as the smallest set of committed identifiers $S \subseteq commit$ such that, for each identifier

Algorithm 3.3: Atlas execution protocol.

```
57 loop
58   let S be the smallest subset of commit such that
       ∀id ∈ S. (dep[id] ⊆ S ∪ execute)
59   for id ∈ S ordered by <
60     execute(cmd[id])
61     phase[id] ← EXECUTE
```

$id \in S$, its dependencies are in the batch or already executed: $\texttt{dep}[id] \subseteq S \cup execute$ (line 58). This ensures that a command can only be executed after its dependencies or in the same batch with them, which yields the following invariant.

Invariant 3.3. Assume two commands c and c' with identifiers id and id', respectively. If a process executes a batch of commands containing c before executing a batch containing c', then $id' \notin \texttt{dep}[id]$.

As processes agree on the dependencies of each command (Invariant 3.1), the batch in which a command is executed is equal in every process, as reflected in following invariant.

Invariant 3.4. If a process executes command c in batch S and another process executes the same command c in batch S', then $S = S'$.

Inside a batch, commands are ordered according to some fixed total order $<$ on identifiers (line 59). This guarantees that conflicting commands are executed in a consistent order across all processes.

Consider again the example in Figure 3.1, where the final dependencies are $\texttt{dep}[\mathtt{a}] = \emptyset$ and $\texttt{dep}[\mathtt{b}] = \{\mathtt{a}\}$. There are two cases, depending on the order in which processes commit the commands $\mathtt{a}$ and $\mathtt{b}$:

- $\mathtt{a}$ then $\mathtt{b}$: at processes 1 and 2. When the command $\mathtt{a}$ is committed, the processes execute it in a singleton batch, as it has no dependencies. When later the command $\mathtt{b}$ is committed, the processes execute it in a singleton batch too, since its only dependency $\mathtt{a}$ has already been executed.

- $\mathtt{b}$ then $\mathtt{a}$: at processes 3, 4 and 5. When the command $\mathtt{b}$ is committed, the processes cannot execute it, as its dependency $\mathtt{a}$ has not yet been committed. When later the command $\mathtt{a}$ is committed, the processes execute two singleton batches: first $\mathtt{a}$, then $\mathtt{b}$.

Note that $\mathtt{a}$ is executed before $\mathtt{b}$ in both cases, thus ensuring a consistent execution order across processes.

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

3.2.4 Properties and Comparison with EPaxos

Complexity. Atlas commits a command after two communication delays when taking the fast path, and four otherwise. As pointed out in §3.2.1, when $f = 1$, a fast quorum contains exactly a majority of processes and Atlas *always* takes the fast path. This is optimal for leaderless protocols [18, 41] and results in a significant performance pay-off (§5).

Fault tolerance. Atlas is parameterized by the number of tolerated concurrent faults f: smaller values of f yield smaller fast and slow quorums, thus reducing latency. As observed in the literature [4, 17] and as we experimentally confirm in §5.1, assuming small values of f is acceptable for geo-distribution. Furthermore, violating our assumption that the number of failures is bounded by f may only compromise the liveness of the protocol, and never its safety: if more than f transient outages occur, due to, e.g., connectivity problems, Atlas will just block until enough sites are reachable.

Comparison with EPaxos. Atlas belongs to the family of leaderless SMR protocols. We now provide a concise comparison with the most prominent protocol in this family, EPaxos [11]. The two protocols share the message flow, including the splitting into fast and slow paths. However, as we demonstrate experimentally in §5, Atlas outperforms EPaxos in several settings, which is due to a number of novel design decisions that we took.

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

Second, Atlas allows choosing the number of failures f independently of the size of the system r, which yields fast quorums of size $\lfloor \frac{r}{2} \rfloor + f$. EPaxos assumes up to $\lfloor \frac{r}{2} \rfloor$ failures and sets the fast quorum size to $\lfloor \frac{3r}{4} \rfloor$. Our decision results in smaller quorums for small values of f, which are appropriate in planet-scale systems [4, 17, 42]; smaller quorums then result in lower latency.

Third, Atlas achieves its smaller fast quorums with a simple recovery protocol that is able to recover fast-path decisions using Property 3.2. In contrast, EPaxos recovery mechanism is much more complex [43, 44], and in fact has been recently shown to contain a bug [45]. Due to this complexity, the authors of Janus (§2.5) based their protocol on a simpler version of EPaxos that employs large fast quorums [15, 43]. Such complexity also does not allow EPaxos to be easily modified to exploit the independent bound on failures f, unlike plain Paxos [35] or Fast Paxos [46].

Lack of liveness. The Atlas execution mechanism presented in §3.2.3 is the same as that of EPaxos [11]. For this reason, Atlas does not ensure liveness even under a synchronous network. We present in §4.2.4 a counterexample that demonstrates this. Tempo, which we present in the next chapter (§4), employs a different execution mechanism that is able to avoid this issue.

3.2.5 Optimizations

This section presents one mechanism employed by the Atlas protocol to accelerate command execution and one mechanism that can be used to avoid slow fast-quorum processes.

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

Avoiding a fast-quorum straggler. Notice that if a process in the fast quorum is slow, Atlas needs to execute recovery. We now sketch a mechanism that can be used to avoid such fast-quorum stragglers. First, instead of sending the MCollect message only to the fast-quorum processes, the initial coordinator sends it to all processes. Similarly to the fast-quorum processes, the remaining processes also propose a set of dependencies for the command. Then, if all fast-quorum processes are responsive, the replies by non-fast-quorum processes can be simply ignored. However, if some fast-path process is slow to answer, these additional replies can be used to take the slow path: this path can always be taken as long as the coordinator has the replies from a majority of processes, thus maintaining Property 3.1.

3.3 Correctness

In this section we prove that the Atlas protocol satisfies the SMR specification (§3.1). We omit the trivial proof of Validity and prove Ordering next.

Consider the auxiliary invariants below:

Invariant 3.5. At any process, if $\mathtt{cmd}[id] \neq \mathtt{noOp}$, then $\mathtt{cmd}[id]$ has been previously submitted by a client.

Invariant 3.6. Assume $\mathtt{MCollect}(id, c, \_, \_)$ has been sent. Then for any $\mathtt{MConsensus}(id, c', \_, \_)$, $\mathtt{MCommit}(id, c', \_)$ and $\mathtt{MRec}(id, c', \_)$, we have $c' = c$ or $c' = \mathtt{noOp}$.

Invariant 3.7. Assume $\mathtt{MConsensus}(id, \_, \_, b)$ has been sent. Then $b = \mathtt{initial}(id)$ or $b > r$.

Invariant 3.8. Assume $\mathtt{MConsensus}(id, c, D, b)$ and $\mathtt{MConsensus}(id, c', D', b')$ have been sent. If $b = b'$, then $c = c'$ and $D = D'$.

Invariant 3.9. Assume $\texttt{MRecAck}(id, c, \_, Q, ab, \_)$ and $\texttt{MRecAck}(id, c', \_, Q', ab', \_)$ have been sent. If $Q \neq \emptyset$ and $Q' \neq \emptyset$, then $Q = Q'$. If additionally $ab = ab' = 0$, then $c = c'$.

Invariant 3.10. Assume $\texttt{MRecAck}(\_, \_, \_, \_, ab, b)$ has been sent by some process. Then $ab < b$.

Invariant 3.11. Assume $\texttt{MConsensusAck}(id, b)$ and $\texttt{MRecAck}(id, \_, \_, \_, ab, b')$ have been sent by some process. If $b' > b$, then $b \leq ab < b'$ and $ab \neq 0$.

Invariant 3.12. Assume a slow quorum has received $\texttt{MConsensus}(id, c, D, b)$ and responded to it with $\texttt{MConsensusAck}(id, b)$. For any $\texttt{MConsensus}(id, c', D', b')$ sent, if $b' > b$, then $c' = c$ and $D' = D$.

Invariant 3.13. Assume $\texttt{MCommit}(id, c, D)$ has been sent at line 16. Then for any $\texttt{MConsensus}(id, c', D', \_)$ sent, $c' = c$ and $D' = D$.

Invariant 3.14. Assume $\texttt{MCommit}(id, c, \_)$ and $\texttt{MCommit}(id', c', \_)$ have been sent, $c \neq \texttt{noOp}$, $c' \neq \texttt{noOp}$ and $\texttt{conflict}(c, c')$. Assume further that some process sends two messages: either $\texttt{MCollectAck}(id, dep)$ or $\texttt{MRecAck}(id, \_, dep, \_, 0, \_)$ and either $\texttt{MCollectAck}(id', dep')$ or $\texttt{MRecAck}(id', \_, dep', \_, 0, \_)$. Then $id' \in dep$ or $id \in dep'$.

Invariants 3.5-3.11 easily follow from the structure of the protocol. Next we prove the rest of the invariants and use them to prove Invariant 3.1 and Invariant 3.2′ (we omit the easy proofs of Invariant 3.3 and Invariant 3.4). Finally, we introduce and prove two lemmas that are then used to prove the Ordering property of the SMR specification.

Proof of Invariant 3.12. Assume that at some point

> (*) a slow quorum has received $\texttt{MConsensus}(id, c, D, b)$ and responded to it with $\texttt{MConsensusAck}(id, b)$.

We prove by induction on b' that, if a process i sends $\texttt{MConsensus}(id, c', D', b')$ with $b' > b$, then $c' = c$ and $D' = D$. Given some b^*, assume this property holds for all $b' < b^*$. We now show that it holds for $b' = b^*$. We make a case split depending on the transition of process i that sends the MConsensus message.

First, assume that process i sends MConsensus at line 18. In this case, $b' = i$. Since $b' > b$, we have $b < i$. But this contradicts Invariant 3.7. Hence, this case is impossible.

The remaining case is when process i sends MConsensus during the transition at line 45. In this case, i has received

$$\texttt{MRecAck}(id, c_j, dep_j, \_, ab_j, b')$$

from all processes j in a recovery quorum Q^R. Let $ab_{\texttt{max}} = \texttt{max}\{ab_j \mid j \in Q^R\}$; then by Invariant 3.10 we have $ab_{\texttt{max}} < b'$.

Since the recovery quorum Q^R has size $r-f$ and the slow quorum from (*) has size $f+1$, we get that at least one process in Q^R must have received the $\texttt{MConsensus}(id, c, D, b)$ message and responded to it with $\texttt{MConsensusAck}(id, b)$. Let one of these processes be p. Since $b' > b$, by Invariant 3.11 we have $ab_p \neq 0$, and thus process i executes line 49. By Invariant 3.11 we also have $b \leq ab_p$ and thus $b \leq ab_{\texttt{max}}$.

Consider an arbitrary process $k \in Q^R$, selected at line 48, such that $ab_k = ab_{\texttt{max}}$. We now prove that $c_k = c$ and $dep_k = D$. If $ab_{\texttt{max}} > b$, then since $ab_{\texttt{max}} < b'$, by induction hypothesis we have $c_k = c$ and $dep_k = D$, as required. If $ab_{\texttt{max}} = b$, then since $ab_{\texttt{max}} \neq 0$, process k has received some $\texttt{MConsensus}(id, \_, \_, ab_{\texttt{max}})$ message. By Invariant 3.8, process k must have received the same $\texttt{MConsensus}(id, c, D, ab_{\texttt{max}})$ received by process p. Upon receiving this message, process k stores c in $\texttt{cmd}$ and D in dep and does not change these values at line 40: $ab_{\texttt{max}} \neq 0$ and thus $\texttt{bal}[id]$ cannot be 0 when the process executes this line. Then process k must have sent $\texttt{MRecAck}(id, c_k, dep_k, \_, ab_{\texttt{max}}, b')$ with $c_k = c$ and $dep_k = D$, which concludes the proof. □

Proof of Invariant 3.13. Assume $\texttt{MCommit}(id, c, D)$ has been sent at line 16. Then, the process that sent this $\texttt{MCommit}$ message must be process $\texttt{initial}(id)$. Moreover, we have that for some fast quorum Q^F such that $\texttt{initial}(id) \in Q^F$:

> (*) every process $j \in Q^F$ has received $\texttt{MCollect}(id, c, Q^F, past)$ and responded with $\texttt{MCollectAck}(id, dep_j)$ such that $D = \bigcup_{Q^F} dep = \bigcup_{Q^F} dep$.

We prove by induction on b that, if a process i sends $\texttt{MConsensus}(id, c', D', b)$, then $c' = c$ and $D' = D$. Given some b^*, assume this property holds for all $b < b^*$. We now show that it holds for $b = b^*$.

First note that process i cannot send $\texttt{MConsensus}$ at line 18, since in this case we would have $i = \texttt{initial}(id)$, and $\texttt{initial}(id)$ took the fast path at line 16. Hence, process i must have sent $\texttt{MConsensus}$ during the transition at line 45. In this case, i has received

$$\texttt{MRecAck}(id, c_j, dep_j, Q_j^0, ab_j, b)$$

from all processes j in a recovery quorum Q^R.

If $\texttt{MConsensus}$ is sent at line 49, then we have $ab_k > 0$ for the process $k \in Q^R$ selected at line 48. In this case, before sending $\texttt{MRecAck}$, process k must have received

$$\texttt{MConsensus}(id, c_k, dep_k, ab_k)$$

with $ab_k < b$. Then by induction hypothesis we have $c' = c_k = c$ and $D' = dep_k = D$. This establishes the required.

If $\texttt{MConsensus}$ is not sent in line 49, then we have $ab_k = 0$ for all processes $k \in Q^R$. In this case, process i sends $\texttt{MConsensus}$ in either line 55 or line 56. Since the recovery quorum Q^R has size $r-f$ and the fast quorum Q^F from (*) has size $\lfloor \frac{r}{2} \rfloor + f$, we have that

(**) at least $\lfloor \frac{r}{2} \rfloor$ processes in Q^R are part of Q^F and thus must have received $\texttt{MCollect}(id, c, Q^F, past)$ and responded to it with $\texttt{MCollectAck}$.

Let process p be one these processes. Due to the assignment at line 43 and the check at line 7, process p must have received $\texttt{MCollect}$ before sending $\texttt{MRecAck}$. Then, since $ab_p = 0$, process p reports the initial fast quorum Q^F and command c, i.e., process p sends $\texttt{MRecAck}(id, c_p, \_, Q_p^0, ab_p, \_)$ with $Q_p^0 = Q^F$ and $c_p = c$. Then $Q_p^0 \neq \emptyset$, so that process i must send $\texttt{MConsensus}$ at line 55.

By Invariant 3.9, and since process p has sent $\texttt{MRecAck}(id, c, \_, Q^F, \_, \_)$, any process k selected in line 50 has $Q_k^0 = Q^F$ and $c_k = c$. For this reason, $c' = c_k = c$, as required. We now show that $D' = D$. Let I be the set of fast-quorum processes that replied to the new coordinator, i.e. $I = Q^R \cap Q^F$. By our assumption, process $\texttt{initial}(id)$ sent an $\texttt{MCommit}(id, c, D)$ at line 16. Then due to line 36, this process would reply to $\texttt{MRec}$ with $\texttt{MCommit}$ instead of $\texttt{MRecAck}$. Hence, $\texttt{initial}(id) \notin I$, and with that, the set I is selected in line 53. By Property 3.2, the fast path proposal $D = \bigcup_{Q^F} dep$ can be recovered by the set union of the dependencies initially reported by any $\lfloor \frac{r}{2} \rfloor$ fast quorum members (excluding the initial coordinator). By (**), and since all processes $k \in I$ have $ab_k = 0$, then all processes in I replied with the dependencies that were reported to the initial coordinator. Thus, by Property 3.2 we have $D = \bigcup_{Q^F} dep = \bigcup_I dep = D'$, which concludes the proof. □

Proof of Invariant 3.1. Consider that $\texttt{MCommit}(id, c, D)$ and $\texttt{MCommit}(id, c', D')$ have been sent. We prove that $c = c'$ and $D = D'$.

Note that, if an $\texttt{MCommit}(id, c, D)$ was sent at line 37, then some process sent an $\texttt{MCommit}(id, c, D)$ at line 16 or line 26. Hence, without loss of generality, we can assume that the two $\texttt{MCommit}$ under consideration were sent at line 16 or at line 26. We can also assume that the two $\texttt{MCommit}$ have been sent by different processes. Only one process can send an $\texttt{MCommit}$ at line 16 and only once. Hence, it is sufficient to only consider the following two cases.

Assume first that both $\texttt{MCommit}$ messages are sent at line 26. Then for some b, a slow quorum has received $\texttt{MConsensus}(id, c, D, b)$ and responded to it with $\texttt{MConsensusAck}(id, b)$. Likewise, for some b', a slow quorum has received $\texttt{MConsensus}(id, c', D', b')$ and responded to it with $\texttt{MConsensusAck}(id, b')$.

Assume without loss of generality that $b \leq b'$. If $b < b'$, then $c' = c$ and $D' = D$ by Invariant 3.12. If $b = b'$, then $c' = c$ and $D' = D$ by Invariant 3.8. Hence, in this case $c' = c$ and $D' = D$, as required.

Assume now that $\texttt{MCommit}(id, c, D)$ was sent at line 16 and $\texttt{MCommit}(id, c', D')$ at line 26. Then for some b, a slow quorum has received $\texttt{MConsensus}(id, c', D', b)$ and responded to it with $\texttt{MConsensusAck}(id, b)$. Then by Invariant 3.13, we must have $c' = c$ and $D' = D$, as required. □

Proof of Invariant 3.14. Assume $\texttt{MCommit}(id, c, \_)$ and $\texttt{MCommit}(id', c', \_)$ have been sent, $c \neq \texttt{noOp}$, $c' \neq \texttt{noOp}$ and $\texttt{conflict}(c, c')$. Assume further that process j sends two messages: either $\texttt{MCollectAck}(id, dep)$ or $\texttt{MRecAck}(id, \_, dep, \_, 0, \_)$ and either $\texttt{MCollectAck}(id', dep')$

or $\texttt{MRecAck}(id', \_, dep', \_, 0, \_)$. If $\texttt{MCollectAck}(id, dep)$ is sent, it must be in response to $\texttt{MCollect}(id, d, \_, \_)$, and by Invariant 3.6 we have $d = c$. Similarly, if $\texttt{MCollectAck}(id', dep')$ is sent, it must be in response to $\texttt{MCollect}(id', d', \_, \_)$, and by Invariant 3.6 we have $d' = c'$. If $\texttt{MRecAck}(id, \_, dep, \_, 0, \_)$ is sent, it must be in response to $\texttt{MRec}(id, d, \_)$, and by Invariant 3.6 we have $d \in \{c, \texttt{noOp}\}$. If $\texttt{MRecAck}(id', \_, dep', \_, 0, \_)$ is sent, it must be in response to $\texttt{MRec}(id', d', \_)$, and by Invariant 3.6 we have $d' \in \{c', \texttt{noOp}\}$.

Without loss of generality, assume that process j sends the message about id before the message about id'. We prove that $id \in dep'$. We have four cases depending on which message ($\texttt{MCollectAck}$ or $\texttt{MRecAck}$) is sent for each identifier:

1) Process j sends $\texttt{MCollectAck}(id, dep)$ and then $\texttt{MCollectAck}(id', dep')$. When handling $\texttt{MCollect}(id, c, \_, \_)$, process j stores c in $\texttt{cmd}[id]$. By Invariant 3.6, $\texttt{cmd}[id]$ can only change to $\texttt{noOp}$. When handling $\texttt{MCollect}(id', c', \_, \_)$, since $\texttt{cmd}[id] \in \{c, \texttt{noOp}\}$ and $\texttt{noOp}$ conflicts with all commands, we have $id \in \mathbf{conflicts}(c')$ in line 8, and thus $id \in dep'$ in $\texttt{MCollectAck}(id', dep')$, as required.

2) Process j sends $\texttt{MCollectAck}(id, dep)$ and then $\texttt{MRecAck}(id', \_, dep', \_, 0, \_)$. When handling $\texttt{MCollect}(id, c, \_, \_)$, process j stores c in $\texttt{cmd}[id]$. By Invariant 3.6, $\texttt{cmd}[id]$ can only change to $\texttt{noOp}$. When handling $\texttt{MRec}(id', d', \_)$ with $d' \in \{c', \texttt{noOp}\}$ we have two cases depending on $\texttt{phase}[id']$. If $id' \in start$, then since $\texttt{cmd}[id] \in \{c, \texttt{noOp}\}$ and $\texttt{noOp}$ conflicts with all commands, we have $id \in \mathbf{conflicts}(d')$ in line 41. If $id' \notin start$, then process j is a member of the original fast quorum for id' and thus included id into $\texttt{dep}[id']$ when it processed $\texttt{MCollect}(id', c', \_, \_)$. Thus, in both cases $id \in dep'$ in $\texttt{MRecAck}(id', \_, dep', \_, 0, \_)$, as required.

3) Process j sends $\texttt{MRecAck}(id, \_, dep, \_, 0, \_)$ and then $\texttt{MCollectAck}(id', dep')$. Analogous to the above.

4) Process j sends $\texttt{MRecAck}(id, \_, dep, \_, 0, \_)$ and then $\texttt{MRecAck}(id', \_, dep', \_, 0, \_)$. Analogous to the above. □

Proof of Invariant 3.2′. Assume that $\texttt{MCommit}(id, c, D)$ and $\texttt{MCommit}(id', c', D')$ have been sent with $id \neq id'$, $c \neq \texttt{noOp}$, $c' \neq \texttt{noOp}$ and $\texttt{conflict}(c, c')$. The protocol structure ensures that $D = \bigcup_Q dep$ for Q and dep given as parameters of handlers at lines 12 or 45, and the computation of D occurs at lines 14 or 54. We start by proving that there exists a quorum $\widehat{Q}$ with $|\widehat{Q}| \geq \lfloor \frac{r}{2} \rfloor + 1$ and $\widehat{dep}$ such that $\bigcup_Q dep = \bigcup_{\widehat{Q}} \widehat{dep}$, where each process $j \in \widehat{Q}$ computes its $\widehat{dep}_j$ in either line 8 or line 41 and sends it in either $\texttt{MCollectAck}(id, \widehat{dep}_j)$ or $\texttt{MRecAck}(id, \_, \widehat{dep}_j, \_, 0, \_)$.

The computation of D occurs either in the transition at line 14 or at line 54. If the computation of D occurs in the transition at line 14, then Q is a fast quorum with size $\lfloor \frac{r}{2} \rfloor + f$. In this case, we let $\widehat{Q} = Q$ and $\widehat{dep} = dep$. Since $f \geq 1$, we have $|\widehat{Q}| \geq \lfloor \frac{r}{2} \rfloor + 1$, as required. If the computation of D occurs at line 54, we have two situations depending on whether $\texttt{initial}(id)$ replies to the new coordinator (line 52). If it does ($s = true$, line 52), then Q is a recovery quorum of size $r - f$. In this case, we let $\widehat{Q} = Q$ and $\widehat{dep} = dep$. Since $f \leq \lfloor \frac{r-1}{2} \rfloor$, we have $|\widehat{Q}| \geq \lfloor \frac{r}{2} \rfloor + 1$, as required. If $\texttt{initial}(id)$

does not reply ($s = \mathit{false}$, line 52), then Q consists of the fast quorum members that are part of the recovery quorum (line 51). Given that fast quorum size is $\lfloor \frac{r}{2} \rfloor + f$ and the recovery quorum size is $r - f$, in this case Q contains at least $\lfloor \frac{r}{2} \rfloor + f - f = \lfloor \frac{r}{2} \rfloor$ fast quorum processes, and thus $|Q| \geq \lfloor \frac{r}{2} \rfloor$. Since D is computed in the branch where the initial fast quorum is known (line 50), at least one of the fast quorum members in Q must have computed its set of dependencies at line 8, including in its dependencies those reported by the original coordinator. In this case, we let $\widehat{Q} = Q \cup \{\mathtt{initial}(id)\}$, $\forall j \in Q.\, \widehat{dep}_j = dep_j$ and $\widehat{dep}_{\mathtt{initial}(id)}$ be the set of dependencies sent by $\mathtt{initial}(id)$ in its $\mathtt{MCollectAck}(id, \widehat{dep}_{\mathtt{initial}(id)})$ message. Since $|Q| \geq \lfloor \frac{r}{2} \rfloor$ and $\mathtt{initial}(id) \notin Q$ (as it did not reply), we have $|\widehat{Q}| \geq \lfloor \frac{r}{2} \rfloor + 1$, as required.

Similarly to the above, we can also prove that there exists a quorum $\widehat{Q'}$ with $|\widehat{Q'}| \geq \lfloor \frac{r}{2} \rfloor + 1$ and $\widehat{dep'}$ such that $\bigcup_{Q'} dep' = \bigcup_{\widehat{Q'}} \widehat{dep'}$, where each process $j \in \widehat{Q'}$ computes its $\widehat{dep'_j}$ in either line 8 or line 41 and sends its $\widehat{dep'_j}$ in either $\mathtt{MCollectAck}(id', \widehat{dep'_j})$ or $\mathtt{MRecAck}(id', \_, \widehat{dep'_j}, \_, 0, \_)$.

We now prove that $id' \in D$ or $id \in D'$. By contradiction, assume that $id' \notin D$ and $id \notin D'$. Since $id' \notin D$, we have $\forall j \in \widehat{Q}.\, id' \notin \widehat{dep}_j$. Similarly, since $id \notin D'$, we have $\forall j \in \widehat{Q'}.\, id \notin \widehat{dep'_j}$. Given that $|\widehat{Q}| \geq \lfloor \frac{r}{2} \rfloor + 1$ and $|\widehat{Q'}| \geq \lfloor \frac{r}{2} \rfloor + 1$, $\widehat{Q}$ and $\widehat{Q'}$ must intersect. For this reason, there must exist a process $p \in \widehat{Q} \cap \widehat{Q'}$ such that $id' \notin \widehat{dep}_p$ and $id \notin \widehat{dep'_p}$. But this contradicts Invariant 3.14. □

We now prove that Atlas ensures Ordering. First we introduce the following two lemmas.

Lemma 3.1. The relation $\bigcup_{i=1}^{r} \mapsto_i$ is asymmetric.

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

Assume first that c and c' are executed at process i in the same batch S. Then by Invariant 3.4 they also have to be executed at process j in the batch S. Since inside a batch commands are ordered using the fixed order $<$ on their identifiers, c and c' have to be executed in the same order at the two processes: a contradiction.

Assume now that c and c' are not executed at process i in the same batch. Then by Invariant 3.4 this also must be the case at process j. Hence, Invariant 3.3 implies that $id' \notin \mathtt{dep}[id]$ at process i, and $id \notin \mathtt{dep}[id']$ at process j. Then process i received $\mathtt{MCommit}(id, c, D)$ with $id' \notin D$, and process j received $\mathtt{MCommit}(id', c', D')$ with $id \notin D'$, which contradicts Invariant 3.2′. □

Lemma 3.2. Assume $c_1 \mapsto \ldots \mapsto c_n$ for $n \geq 2$. Whenever a process i executes c_n, some process has already executed c_1.

Proof. We prove the lemma by induction on n. The base case of $n = 2$ directly follows from the definition of $\mapsto$. Take $n > 3$ and assume $c_1 \mapsto \ldots \mapsto c_{n-1} \mapsto c_n$. Consider the moment when a process i executes c_n. We want to show that by this moment some process has already executed c_1.

Since $c_{n-1} \mapsto c_n$, either $c_{n-1} \rightsquigarrow c_n$ or $c_{n-1} \mapsto_j c_n$ for some process j. Consider first the case when $c_{n-1} \rightsquigarrow c_n$. Then c_{n-1} is executed at some process k before c_n is submitted and, hence, before c_n is executed at process i. By induction hypothesis, c_1 is executed at some process before c_{n-1} is executed at process k and, hence, before c_n is executed at process i, as required. We now consider the case when $c_{n-1} \mapsto_j c_n$ for some process j. Since process i executes c_n, we must have either $c_{n-1} \mapsto_i c_n$ or $c_n \mapsto_i c_{n-1}$. The latter case would contradict Lemma 3.1, so that $c_{n-1} \mapsto_i c_n$. By induction hypothesis, c_1 is executed at some process before c_{n-1} is executed at process i and, hence, before c_n is executed at process i, as required. □

Proof of Ordering. By contradiction, assume that $c_1 \mapsto \ldots \mapsto c_n = c_1$ for $n \geq 2$. Then some process executed c_1. Consider the moment when the first process did so. By Lemma 3.2 some process has already executed c_1 before this, which yields a contradiction. □

3.4 Summary and Related Work

The classical way of implementing SMR is by funnelling all commands through a single leader replica [7, 8, 47, 48], which can create a bottleneck. A way to mitigate this problem is to distribute the leader responsibilities round-robin among replicas, as done in Mencius [20]. However, this makes the system run at the speed of the slowest replica.

Exploiting commutativity to improve the scalability of SMR was first proposed in Generalized Paxos [13] and Generic Broadcast [14]. These protocols still rely on a leader to order concurrent non-commuting commands, which also creates a bottleneck.

The closest SMR protocol to `Atlas` is EPaxos [11], which is also leaderless and exploits commutativity. We compared `Atlas` with EPaxos in detail in §3.2.4. There have been two follow-up protocols to EPaxos, Alvin [12] and Caesar [10]. `Atlas` compares to these protocols similarly to EPaxos; in particular, both follow-ups have large fast quorums that depend on the overall number of processes only.

Flexible Paxos [35] reduces the size of Paxos Phase 2 quorums to $f + 1$, a technique we also use on the slow path of `Atlas`. However, this technique is not directly applicable to computing dependencies via fast path, as required by leaderless SMR. To the best of our knowledge, `Atlas` is the first protocol to reduce the size of fast quorums to $\lfloor \frac{r}{2} \rfloor + f$.

An approach to scaling SMR is to shard the state of the application being replicated and add cross-shard coordination to preserve consistency [49]. Such approaches build on a non-sharded SMR protocol and are hence orthogonal to our proposal: `Atlas` can be combined with them to scale SMR even further. Protocols such as M2Paxos [50], WPaxos [51] and DPaxos [52] scale up SMR using a variation of the sharding approach. These protocols exploit access locality by optimizing for workloads where commands do not frequently access objects in multiple locations.

There have been recent proposals of SMR protocols that improve scalability using special hardware capabilities, such as low-latency switches or RDMA [53–55]. However, currently these protocols work within a single data center only.

Summary This chapter presented Atlas, the first leaderless SMR protocol parameterized with the number of allowed failures. Atlas is designed for planet-scale systems where concurrent site failures are rare. It employs three mechanisms to reduce latency: small quorums, a flexible fast-path condition, and non-fault-tolerant reads. Atlas leverages the same execution mechanism of EPaxos based on explicit dependencies. As highlighted in §2, this mechanism can result in pathological scenarios where the protocol continuously commits commands but can never execute them, even under a synchronous network [11, 19]. In practice, this translates into a high tail latency (§5). In the next chapter we introduce Tempo, a leaderless protocol that leverages the ideas of Atlas presented in this chapter, but addresses these liveness and performance issues.

4

TEMPO: Predictable Leaderless Consensus

In this chapter we present Tempo, a leaderless SMR protocol that offers *predictable performance*. Existing leaderless SMR protocols suffer from drawbacks in the way they order commands. For example, EPaxos and Atlas 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 (§2). In practice, their performance is unpredictable, and in particular, exhibits a high tail latency [9, 19]. These drawbacks carry over to the setting of partial replication where they are aggravated by the fact that commands span multiple machines. To lift these limitations, Tempo redesigns the way commands are typically ordered in leaderless SMR. Tempo assigns a scalar *timestamp* to each command and executes 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 [10, 21–23]. 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. This allows Tempo to offer low tail latency even in contended workloads, thus ensuring predictable performance. Tempo builds on Atlas and thus it also offers *low (average) latency* and *simple recovery* (§3).

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 organize this chapter as follows. We start by covering the system model in §4.1. This is followed by a detailed explanation of the Tempo protocol in §4.2: we cover the single-partition commit protocol in §4.2.1, the execution protocol in §4.2.2, a comparison with Atlas, EPaxos and Caesar in §4.2.3, the multi-partition commit protocol in §4.2.5, the recovery protocol in §4.2.6, the liveness protocol in §4.2.7, and a few optimizations in §4.2.9. In §4.3 we show the correctness of Tempo. We conclude this chapter with a summary and related work in §4.4.

4.1 System Model

We consider a general version of State-Machine Replication (SMR) (§3.1) where each machine replicates only a part of the service state – *partial SMR (PSMR)* [15, 56, 57]. 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, where f can be any value such that $1 \leq f \leq \lfloor \frac{r-1}{2} \rfloor$. We write $\mathbb{I}_p$ for the set of all the processes replicating a partition p, $\mathbb{I}_c$ for the set of processes that replicate the partitions accessed by a command c, and $\mathbb{I}$ for the set of all processes.

A *PSMR protocol* provides a command $\texttt{submit}(c)$, which allows a process i to submit a command $c \in \mathcal{C}$ 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: $i \in \mathbb{I}_c$. For each partition p accessed by c, the protocol then triggers an upcall $\texttt{execute}_p(c)$ 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* [6] – 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 $c \mapsto_i d$ when they access a common partition and c is executed before d at some process $i \in \mathbb{I}_c \cap \mathbb{I}_d$. We also define the following *real-time order*: $c \rightsquigarrow d$ if the command c returns before the command d was submitted. Let $\mapsto = (\bigcup_{i \in \mathbb{I}} \mapsto_i) \cup \rightsquigarrow$. 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 $\mapsto$ 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 $\mathbb{I}_c$.

The Ordering property ensures that commands are executed in a consistent manner throughout the system [58]. 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 (§2.1). In the following sections we present `Tempo`, which satisfies the above specification.

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 [15, 59, 60]. It can also be used to construct general-purpose transactions [15, 61].

Note that SMR is a special case of PSMR where each machine replicates the whole service state. In this case, however, the above PSMR specification is not equivalent to the SMR specification introduced in §3.1. The SMR specification in §3.1 is parameterized by a `conflict` relation which only prevents two commands c and d from being executed in contradictory orders by different processes when $\texttt{conflict}(c, d)$. In particular, it allows two commands reading the same partition to be executed in contradictory orders (since read commands do not `conflict` with each other). On the other hand, the PSMR specification above enforces that any two commands accessing the same partition are executed in the the same order at all the processes replicating such partition, even when the commands do not `conflict`. We discuss the implications of this assumption at the end of §4.2.3. Lastly, the SMR specification in §3.1 does not include a Liveness property as the PSMR specification above. This is due to the fact that `Tempo` ensures Liveness, while `Atlas` does not.

4.2 The `Tempo` Protocol

For simplicity, we first present the protocol in the case when there is only a single partition p, and cover the general case in §4.2.5. 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 (§4.2.6). `Tempo` ensures that, even in

case of failures, processes agree on the timestamp assigned to the command, as stated by the following invariant.

Invariant 4.1 (Timestamp agreement)**.** Two processes cannot commit the same command with different timestamps.

A coordinator computes a timestamp for a command as follows (§4.2.1). It first forwards the command to a *fast quorum* of $\lfloor \frac{r}{2} \rfloor + f$ processes, including the coordinator itself. Each process maintains a $\texttt{Clock}_p$ variable. When the process receives a command from the coordinator, it increments $\texttt{Clock}_p$ and replies to the coordinator with the new $\texttt{Clock}_p$ 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 Invariant 4.1. 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 [35]. 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.

Invariant 4.2 (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 (§4.2.2), each process i tracks timestamp proposals issued by other processes. Once the clocks ($\texttt{Clock}_p$) 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.

4.2.1 Commit Protocol

Algorithm 4.1 specifies the Tempo 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 $\texttt{submit}(c)$ at a process i that replicates a partition accessed by the command (line 1). Process i then creates a unique identifier $id \in \mathcal{D}$ 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,

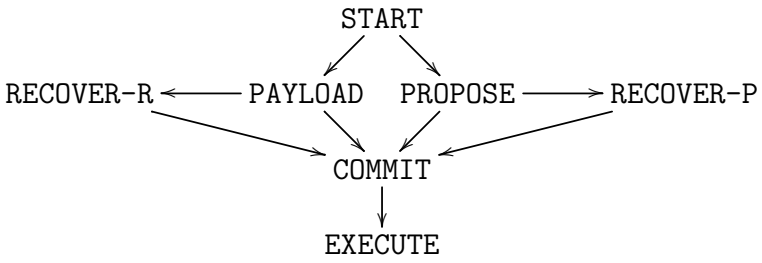


Figure 4.1: Command journey through phases in Tempo.

in what follows Q contains only one fast quorum, $Q[p]$. Finally, process i sends $\texttt{MSubmit}(id, c, Q)$ to a set of processes $\mathbb{I}_c^i$, which in the single-partition case simply denotes $\{i\}$.

A command goes through several *phases* at each process: from the initial phase START, to a COMMIT phase once the command is committed, and an EXECUTE phase once it is executed. We summarize these phases and allowed phase transitions in Figure 4.1. A mapping phase at a process tracks the progress of a command with a given identifier through phases. As before, the name of the phase written in lower case denotes all the commands in that phase, e.g., $start = \{id \in \mathcal{D} \mid \texttt{phase}[id] = \texttt{START}\}$. We also define $pending$ as follows: $pending = payload \cup propose \cup recoverp \cup recoverr$. We summarize in Table 4.1 the data maintained by each process for a command with identifier id, where $\mathcal{O}$ denotes the set of all partitions. Note that Table 4.1 also contains data that is not specific to any command, such as $\texttt{Clock}_p$.

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 $\texttt{Clock}_p + 1$. After computing the proposal, the coordinator sends an MPropose 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 MPropose message to itself. As mentioned earlier, self-addressed messages are delivered immediately.

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 (§4.2.6).

Propose phase

Upon receiving an MPropose 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

Algorithm 4.1: Tempo commit protocol at process $i \in \mathbb{I}_p$.

```
1  submit(c)
2      pre: i ∈ I_c
3      id ← next_id(); Q ← fast_quorums(i, I_c)
4      send MSubmit(id, c, Q) to I_c^i
5  receive MSubmit(id, c, Q)
6      t ← Clock_p + 1
7      send MPropose(id, c, Q, t) to Q[p]
8      send MPayload(id, c, Q) to I_p \ Q[p]
9  receive MPayload(id, c, Q)
10     pre: id ∈ start
11     cmd[id] ← c; quorums[id] ← Q; phase[id] ← PAYLOAD
12 receive MPropose(id, c, Q, t) from j
13     pre: id ∈ start
14     cmd[id] ← c; quorums[id] ← Q; phase[id] ← PROPOSE
15     ts[id] ← proposal(id, t)
16     send MProposeAck(id, ts[id]) to j
17     send MBump(id, ts[id]) to I_c^i
18 receive MBump(id, t)
19     pre: id ∈ propose
20     bump(t)
21 receive MProposeAck(id, t_j) from ∀ j ∈ Q
22     pre: id ∈ propose ∧ Q = quorums[id][p]
23     t ← max{t_j | j ∈ Q}
24     if count(t) ≥ f then send MCommit(id, t) to I_cmd[id]
25     else send MConsensus(id, t, i) to I_p
26 receive MCommit(id, t_j) from j ∈ P s.t. P contains one process from each partition accessed by cmd[id]
27     pre: id ∈ pending
28     ts[id] ← max{t_j | j ∈ P}; phase[id] ← COMMIT; bump(ts[id])
29 receive MConsensus(id, t, b) from j
30     pre: bal[id] ≤ b
31     ts[id] ← t; bal[id] ← b; abal[id] ← b; bump(t)
32     send MConsensusAck(id, b) to j
33 receive MConsensusAck(id, b) from Q
34     pre: bal[id] = b ∧ |Q| = f + 1
35     send MCommit(id, ts[id]) to I_cmd[id]
36 proposal(id, m)
37     t ← max(m, Clock_p + 1)
38     Detached ← Detached ∪ {⟨i, u⟩ | Clock_p + 1 ≤ u ≤ t − 1}
39     Attached[id] ← {⟨i, t⟩}
40     Clock_p ← t
41     return t
42 bump(t)
43     t ← max(t, Clock_p)
44     Detached ← Detached ∪ {⟨i, u⟩ | Clock_p + 1 ≤ u ≤ t}
45     Clock_p ← t
```

Table 4.1: Tempo variables at a process from partition p.

$\texttt{cmd}[id] \leftarrow \bot$	$\in \mathcal{C}$	Command
$\texttt{ts}[id] \leftarrow 0$	$\in \mathbb{N}$	Timestamp
$\texttt{phase}[id] \leftarrow \texttt{START}$		Phase
$\texttt{quorums}[id] \leftarrow \emptyset$	$\in \mathcal{O} \hookrightarrow \mathcal{P}(\mathbb{I}_p)$	Fast quorum used per partition
$\texttt{bal}[id] \leftarrow 0$	$\in \mathbb{N}$	Current ballot
$\texttt{abal}[id] \leftarrow 0$	$\in \mathbb{N}$	Last accepted ballot
$\texttt{Clock}_p \leftarrow 0$	$\in \mathbb{N}$	Current clock
$\texttt{Detached} \leftarrow \emptyset$	$\in \mathcal{P}(\mathbb{I}_p \times \mathbb{N})$	Detached promises
$\texttt{Attached}[id] \leftarrow \emptyset$	$\in \mathcal{P}(\mathbb{I}_p \times \mathbb{N})$	Attached promises
$\texttt{Promises} \leftarrow \emptyset$	$\in \mathcal{P}(\mathbb{I}_p \times \mathbb{N})$	Known promises

using the function **proposal** and stores it in a mapping `ts`. Finally, the process replies to the coordinator with an `MProposeAck` message, carrying the computed timestamp proposal (line 16). Lines 17-20 are explained later as they are only necessary in the multi-partition case (§4.2.5).

The function **proposal** takes as input an identifier id and a timestamp m and computes a timestamp proposal as $t = \max(m, \texttt{Clock}_p + 1)$, so that $t \geq m$ (line 37). The function *bumps* the $\texttt{Clock}_p$ to the computed timestamp t and returns t (lines 40-41); we explain lines 38-39 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 $\texttt{Clock}_p + 1$ 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 `MProposeAck` message from all the processes in the fast quorum $Q = \mathcal{Q}[p]$ (line 21), 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 24) or the slow path (line 25). Both paths end with the coordinator sending an `MCommit` message containing the command's timestamp. Since $|Q| = \lfloor \frac{r}{2} \rfloor + f$ 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 4.1. For any message $\texttt{MCommit}(id, t)$, there is a set of processes Q such that $|Q| \geq \lfloor \frac{r}{2} \rfloor + 1$ and $t = \max\{t_j \mid j \in Q\}$, where t_j is the output of function $\textbf{proposal}(id, \_)$ 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 (§4.2.6).

Once a process receives an MCommit message (line 26), in the single-partition case it simply saves the command's timestamp received in $\texttt{ts}[id]$. It then moves the command to the COMMIT phase and bumps the $\texttt{Clock}_p$ to the committed timestamp using a function **bump** (line 42). 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 $\texttt{count}(t) \geq f$ in line 24, where $\texttt{count}(t) = |\{j \in Q \mid t_j = t\}|$. If the condition holds, the coordinator immediately sends an MCommit message with the computed timestamp[1]. The protocol ensures that, if the coordinator fails before sending all the MCommit messages, t can be recovered as follows. First, the condition $\texttt{count}(t) \geq f$ 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 $(\lfloor \frac{r}{2} \rfloor + f) - f = \lfloor \frac{r}{2} \rfloor$ quorum members. This observation is captured by the following property.

Property 4.2. Any timestamp committed on the fast path can be obtained by selecting the highest proposal sent in MPropose by at least $\lfloor \frac{r}{2} \rfloor$ fast-quorum processes distinct from the initial coordinator.

This property is identical to Property 3.2 from Atlas (§3). For this reason, the recovery mechanism of the two protocols is very similar, as we detail in §4.2.6.

Fast path examples

Table 4.2 contains several examples that illustrate the fast-path condition of Tempo and Property 4.2. All examples consider $r = 5$ processes while tolerating f faults. We highlight timestamp proposals in bold. Process A acts as the coordinator and sends $\mathbf{6}$ in its MPropose message. The fast quorum Q is $\{\texttt{A}, \texttt{B}, \texttt{C}\}$ when $f = 1$ and $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$ when $f = 2$. The example in Table 4.2 a) considers Tempo $f = 2$. Once process B receives the MPropose with timestamp $\mathbf{6}$, it bumps its $\texttt{Clock}_p$ from 6 to 7 and sends a proposal 7 in the MProposeAck. Similarly, processes C and D bump their $\texttt{Clock}_p$ from 10 to $\mathbf{11}$ and propose $\mathbf{11}$. Thus, A receives proposals $t_\texttt{A} = \mathbf{6}$, $t_\texttt{B} = 7$, $t_\texttt{C} = \mathbf{11}$ and $t_\texttt{D} = \mathbf{11}$, and computes the command's timestamp as $t = \max\{6, 7, 11\} = 11$. Since $\texttt{count}(11) = 2 \geq f$, the coordinator takes the fast path, even though the proposals did not match. In order to understand why this

[1] In line 24 we send the message to $\mathbb{I}_c$ even though this set is equal to $\mathbb{I}_p$ in the single-partition case. We do this to reuse the pseudocode when presenting the multi-partition protocol in §4.2.5.

Table 4.2: 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.

	A	B	C	D	match	fast path
a) $f = 2$	**6**	$6 \rightarrow 7$	$10 \rightarrow \mathbf{11}$	$10 \rightarrow \mathbf{11}$	✗	✓
b) $f = 2$	**6**	$6 \rightarrow 7$	$10 \rightarrow \mathbf{11}$	$5 \rightarrow \mathbf{6}$	✗	✗
c) $f = 1$	**6**	$6 \rightarrow 7$	$10 \rightarrow \mathbf{11}$		✗	✓
d) $f = 1$	**6**	$5 \rightarrow \mathbf{6}$	$1 \rightarrow \mathbf{6}$		✓	✓

is safe, assume that the coordinator fails (before sending all the MCommit messages) along with another fast-quorum process. Independently of which $\lfloor \frac{r}{2} \rfloor = 2$ fast-quorum processes survive ($\{\mathtt{B}, \mathtt{C}\}$ or $\{\mathtt{B}, \mathtt{D}\}$ or $\{\mathtt{C}, \mathtt{D}\}$), timestamp 11 is always present and can be recovered as stated by Property 4.2. This is not the case for the example in Table 4.2 *b*). Here A receives $t_{\mathtt{A}} = \mathbf{6}$, $t_{\mathtt{B}} = 7$, $t_{\mathtt{C}} = \mathbf{11}$ and $t_{\mathtt{D}} = \mathbf{6}$, and again computes $t = \mathtt{max}\{6, 7, 11\} = 11$. Since $\mathtt{count}(11) = 1 < f$, 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 4.2 *c*) and *d*) consider $f = 1$, and the fast path is taken in both, independently of the timestamps proposed. This is because Tempo fast-path condition $\mathtt{count}(\mathtt{max}\{t_j \mid j \in Q\}) \geq f$ trivially holds with $f = 1$. Thus, just like Atlas $f = 1$ (§3), Tempo $f = 1$ always takes the fast path.

Note that when the $\mathtt{Clock}_p$ at a fast-quorum process is below the proposal m sent by the coordinator, i.e., $\mathtt{Clock}_p < m$, the process makes the same proposal as the coordinator. This is not the case when $\mathtt{Clock}_p \geq m$, 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 4.2.

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 Invariant 4.1 in this case, the coordinator first reaches an agreement on the computed timestamp with other processes replicating the same partition. Following the approach of Atlas (§3), this is implemented using single-decree Flexible Paxos [35]. 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 $\mathtt{bal}[id]$ it is currently participating in and the last ballot $\mathtt{abal}[id]$ 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 $\mathtt{bal}[id]$ is not greater than the ballot in the message (line 30). Then it stores the consensus proposal, sets $\mathtt{bal}[id]$ and $\mathtt{abal}[id]$ to the ballot in the message, and replies

to the coordinator with MConsensusAck. Once the coordinator gathers $f + 1$ such replies (line 33), 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.

4.2.2 Execution Protocol

A process executes committed commands in the timestamp order. To this end, as required by Invariant 4.2, a process executes a command only after its timestamp becomes stable, i.e., all commands with a lower timestamp are known. To detect stability, Tempo tracks which timestamp ranges have been used up by each process using the following mechanism.

Promise collection

A *promise* is a pair $\langle j, u \rangle \subseteq \mathbb{I}_p \times \mathbb{N}$ where j is a process and u a timestamp. Promises can be *attached* to some command or *detached*. A promise $\langle j, u \rangle$ 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 $\langle j, u \rangle$ 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 36). This function generates a single attached promise for the proposal t, stored in a mapping Attached (line 39). The function also generates detached promises for the timestamps ranging from $\texttt{Clock}_p + 1$ up to $t - 1$ (line 38): since the process bumps the $\texttt{Clock}_p$ to t (line 40), it will never assign a timestamp in this range. Detached promises are accumulated in the Detached set. In Table 4.2 *d*), process B generates an attached promise $\langle \texttt{B}, \mathbf{6} \rangle$, while C generates $\langle \texttt{C}, \mathbf{6} \rangle$. Process B does not issue detached promises, since its $\texttt{Clock}_p$ is bumped only by 1, from 5 to **6**. However, process C bumps its $\texttt{Clock}_p$ by 5, from 1 to **6**, generating four detached promises: $\langle \texttt{C}, 2 \rangle$, $\langle \texttt{C}, 3 \rangle$, $\langle \texttt{C}, 4 \rangle$, $\langle \texttt{C}, 5 \rangle$.

Algorithm 4.2 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 47)[2]. When a process receives the promises (line 48), 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 49). Lines 51-52 are necessary for the liveness protocol (§4.2.7).

Stability detection

Tempo determines when a timestamp is stable (Invariant 4.2) according to the following theorem.

Theorem 4.1. 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 $|Q| \geq \lfloor \frac{r}{2} \rfloor + 1$.

[2]To minimize the size of these messages, a promise is sent only once in the absence of failures. Promises can be garbage-collected as soon as they are received by all the processes within the partition.

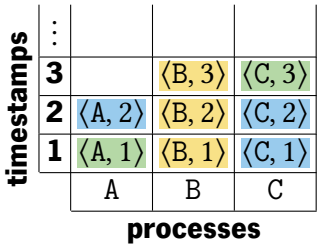
Algorithm 4.2: Tempo execution protocol at process $i \in \mathbb{I}_p$.

```
46 periodically
47   send MPromises(Detached, Attached) to I_p
48 receive MPromises(D, A)
49   C ← ⋃{a | ⟨id, a⟩ ∈ A ∧ id ∈ commit ∪ execute}
50   Promises ← Promises ∪ D ∪ C
51   for ⟨id, _⟩ ∈ A · id ∉ commit ∪ execute
52     send MCommitRequest(id) to I_cmd[id]
53 periodically
54   h ← sort{highest_contiguous_promise(j) | j ∈ I_p}
55   ids ← {id ∈ commit | ts[id] ≤ h[⌊r/2⌋]}
56   for id ∈ ids ordered by ⟨ts[id], id⟩
57     send MStable(id) to I_cmd[id]
58     wait receive MStable(id) from ∀j ∈ I^i_cmd[id]
59     execute_p(cmd[id])
60     phase[id] ← EXECUTE
61 highest_contiguous_promise(j)
62   return max{c ∈ ℕ | ∀u ∈ {1...c} · ⟨j, u⟩ ∈ Promises}
```

Proof. Assume that at some time τ the variable `Promises` at a process i contains all the promises up to s by some set of processes Q with $|Q| \geq \lfloor \frac{r}{2} \rfloor + 1$. Assume further that a command c with identifier id is eventually committed with timestamp $t \leq s$ at some process j, i.e., j receives an $\texttt{MCommit}(id, t)$. We need to show that command c is committed at i at time τ. By Property 4.1 we have $t = \max\{t_k \mid k \in Q'\}$, where $|Q'| \geq \lfloor \frac{r}{2} \rfloor + 1$ and t_k is the output of function $\mathbf{proposal}(id, \_)$ 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 $\langle l, t_l \rangle$ to c (line 39) 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 $\langle l, t_l \rangle$. According to line 49, 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 54). It determines the highest stable timestamp according to Theorem 4.1 as the one at index $\lfloor \frac{r}{2} \rfloor$ 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. Lines 57-58 are explained later as they are only necessary in the multi-partition case (§4.2.5).

To gain more intuition about the above mechanism, consider Figure 4.2, 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 $\langle \texttt{A}, 2 \rangle$ 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 4.2 shows the highest stable timestamp if all the promises in the combination



$X = \{\langle A, 1\rangle, \langle C, 3\rangle\} \longrightarrow 0$
$Y = \{\langle B, 1\rangle, \langle B, 2\rangle, \langle B, 3\rangle\} \longrightarrow 0$
$Z = \{\langle A, 2\rangle, \langle C, 1\rangle, \langle C, 2\rangle\} \longrightarrow 0$
$X \cup Y \longrightarrow 1$
$X \cup Z$ and $Y \cup Z \longrightarrow 2$
$X \cup Y \cup Z \longrightarrow 3$

Figure 4.2: Stable timestamps for different sets of promises.

are in `Promises`. For instance, assume that `Promises` $= Y \cup Z$, 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 49). Moreover, to get committed, c must generate attached promises at a majority of processes (Property 4.1), 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 36) are piggybacked on the `MProposeAck` message, 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 28). This helps ensuring the liveness of the execution mechanism, since the propagation of these promises contributes to advancing the highest stable timestamp.

4.2.3 Timestamp Stability vs Explicit Dependencies

Prior leaderless protocols [9–11, 62] (including `Atlas`) commit each command c with a set of *explicit dependencies* $\texttt{dep}[c]$. 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 4.3 illustrates this issue using four commands x_1, x_2, x_3, x_4 and $r = 3$ processes. Process A submits x_1 and x_4, B submits x_2, and C submits x_3. Commands arrive at the processes in the following order: x_1, x_4, x_3 at A; x_2, x_1 at B; and x_3, x_2 at C. Because in this example only process A has seen command x_4, this command is not yet committed. In `Tempo`, the above command arrival order generates the following attached promises: $\{\langle A, 1\rangle, \langle B, 2\rangle\}$ for x_1, $\{\langle A, 2\rangle\}$ for x_4, $\{\langle B, 1\rangle, \langle C, 2\rangle\}$ for x_2, and $\{\langle C, 1\rangle, \langle A, 3\rangle\}$ for x_3. Commands x_1, x_2 and x_3 are then committed with the following timestamps: $\texttt{ts}[x_1] = 2$, $\texttt{ts}[x_2] = 2$, and $\texttt{ts}[x_3] = 3$. On the left of Figure 4.3 we present the `Promises` variable

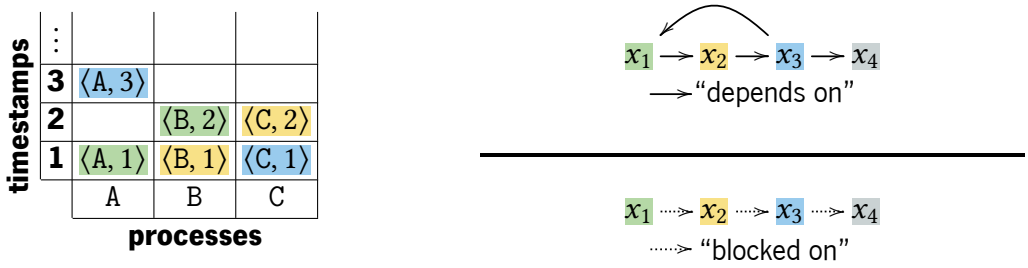


Figure 4.3: Comparison between timestamp stability (left) and two approaches using explicit dependencies (right).

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_4 is not committed, timestamp stability ensures that its timestamp must be greater than 2. Thus, commands x_1 and x_2, 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 [11], BPaxos [62], Gryff [9] and `Atlas` (§3) order commands based on their committed dependencies. For example, in EPaxos the above command arrival order results in commands x_1, x_2 and x_3 committed with the following dependencies: $\mathtt{dep}[x_1] = \{x_2\}$, $\mathtt{dep}[x_2] = \{x_3\}$, $\mathtt{dep}[x_3] = \{x_1, x_4\}$. These form the dependency graph shown on the top right of Figure 4.3. Since the dependency graph may be cyclic (as in Figure 4.3), 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 (§3.2.3). The size of such components is a priori unbounded. In fact, as we show next in §4.2.4, there are pathological scenarios where the protocol continuously commits commands but can never execute them, even under a synchronous network [11, 19]. It may also significantly delay the execution of committed commands, as illustrated by our example: since command x_4 has not yet been committed, and the strongly connected component formed by the committed commands x_1, x_2 and x_3 depends on x_4, no command can be executed – unlike in `Tempo`. As we demonstrate in our experiments (§5), execution delays in such situations lead to high tail latencies.

Dependency-based stability

Caesar [10] associates each command c not only with a set of dependencies $\mathtt{dep}[c]$, but also with a unique timestamp $\mathtt{ts}[c]$. 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 $\mathtt{ts}[c] < \mathtt{ts}[c']$, then $c \in \mathtt{dep}[c']$. 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 x_1 and x_4 with timestamps 1 and 4, respectively, B proposes x_2 with 2, and C proposes x_3 with 3. When B receives command x_1 with timestamp proposal 1, it has already proposed x_2 with timestamp 2. If these proposals succeed and are committed, the above invariant is maintained only if x_1 is a dependency of x_2. However, because x_2 has not yet been committed, its dependencies are unknown and thus B cannot yet ensure that x_1 is a dependency of x_2. For this reason, B must block its response about x_1 until x_2 is committed. Similarly, command x_2 is blocked at C waiting for x_3, and x_3 is blocked at A waiting for x_4. This situation, depicted in the bottom right of Figure 4.3, results in no command being committed – again, unlike in Tempo. In fact, as we show next in §4.2.4, 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 (§5). 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 (§5).

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 (§4.1), so that its performance is not affected by the ratio of reads in the workload. However, this may result in higher average latencies in certain workloads. We show in §5 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. We discuss in §6 how this limitation can be lifted.

4.2.4 Pathological Scenarios

As previously discussed, EPaxos, Caesar and Atlas cannot ensure liveness even under a synchronous network. In this section we present a pathological scenario that demonstrates this. This scenario is a continuation of the example presented in the previous section (Figure 4.3 from §4.2.3) and applies to the three protocols.

Example. Let us consider 3 processes: A, B, C. Assume that all the commands in the example are conflicting. The (infinite) example we consider is as follows:

- A proposes x_1, x_4, x_7, ...
- B proposes x_2, x_5, x_8, ...

- C proposes x_3, x_6, x_9, ...

Processes receive these commands in the following order:

$$\begin{array}{lccccccc} \texttt{A}: & x_1 & & x_4 & x_3 & x_7 & x_6 & \dots \\ \texttt{B}: & x_2 & x_1 & x_5 & x_4 & x_8 & x_7 & \dots \\ \texttt{C}: & x_3 & x_2 & x_6 & x_5 & x_9 & x_8 & \dots \end{array}$$

EPaxos and `Atlas`. In EPaxos and `Atlas`, the command arrival order from above results in the following dependency sets being committed:

- $\texttt{dep}[x_1] = \{\mathbf{x_2}\}$
- $\texttt{dep}[x_2] = \{\mathbf{x_3}\}$
- $\texttt{dep}[x_3] = \{x_1, \mathbf{x_4}\}$
- $\texttt{dep}[x_4] = \{x_1, x_2, \mathbf{x_5}\}$
- $\texttt{dep}[x_5] = \{x_2, x_3, \mathbf{x_6}\}$
- $\texttt{dep}[x_6] = \{x_1, x_3, x_4, \mathbf{x_7}\}$
- ...

With these committed dependencies, no strongly connected component is ever formed. This is because x_1 depends on x_2, which depends on x_3, which in turn depends on x_4, and so on. As a result, commands are continuously committed but never executed.

Caesar. In Caesar, timestamps proposed by processes must be unique. This can be ensured e.g. by assigning timestamps to processes round-robin. For this, assume that in the example above if a process proposes a command x_t, then this command is proposed with timestamp t.

First, process A proposes x_1. When command x_1 reaches process B, B has already proposed x_2. Since x_2 was proposed with a timestamp higher than the timestamp proposed for x_1 (i.e., $2 > 1$), the reply about x_1 is blocked on x_2 until x_2 is committed. However, when x_2 reaches C, C has already proposed x_3, and thus command x_2 is blocked on x_3. This keeps going forever, as depicted in the diagram below (where $\leftarrow$ denotes "blocked on").

$$\begin{array}{lcccc} \texttt{A}: & x_1 & x_4 \leftarrow x_3 & x_7 \leftarrow x_6 & \dots \\ \texttt{B}: & x_2 \leftarrow x_1 & x_5 \leftarrow x_4 & x_8 \leftarrow x_7 & \dots \\ \texttt{C}: & x_3 \leftarrow x_2 & x_6 \leftarrow x_5 & x_9 \leftarrow x_8 & \dots \end{array}$$

As a result, no command is ever committed. This liveness issue is due to Caesar's wait condition [10].

4.2.5 Multi-Partition Protocol

We now explain how the single-partition protocol can be extended to handle commands that access multiple partitions. This is achieved by submitting a multi-partition command at each of the partitions it accesses. 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 $\mathbb{I}_c^i$. For each partition p accessed by c, the set $\mathbb{I}_c^i$ contains a responsive replica of p close to i (e.g., located in the same data center). The processes in $\mathbb{I}_c^i$ then serve as coordinators of c in the respective partitions, following the steps in Algorithm 4.1. This algorithm ends with the coordinator in each partition sending an `MCommit` message to $\mathbb{I}_c$, i.e., all processes that replicate a partition accessed by c (lines 24 and 35; note that $\mathbb{I}_c \neq \mathbb{I}_p$ because c accesses multiple partitions). Hence, each process in $\mathbb{I}_c$ receives as many `MCommits` as the number of partitions accessed by c. Once this happens (line 26), the process computes the final timestamp of the multi-partition command as the highest of the timestamps committed at each partition (line 28), moves the command to the `COMMIT` phase and bumps the $\texttt{Clock}_p$ to the computed timestamp, generating detached promises.

Similarly to the single-partition case, commands are executed using the handler at line 53. This detects command stability using Theorem 4.1, 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 57). Once such a message is received from all the partitions[3] accessed by c, the command is executed. The exchange of `MStable` messages follows the approach in [49] and ensures the real-time order constraint in the Ordering property of PSMR (§4.1). Figure 1 in [49] contains an example (that also applies to `Tempo`) showing why this message exchange is necessary.

Example

Figure 4.4 shows an example of `Tempo` $f = 1$ with $r = 5$ and 2 partitions. Only 3 processes per partition are depicted. Partition x is replicated at `A`, `B` and `C`, and partition y 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 x, `A` computes $\mathbf{6}$ as its timestamp proposal and sends it in an `MPropose` message to the fast quorum $\{\texttt{A}, \texttt{B}, \texttt{C}\}$ (the downward arrows in Figure 4.4). These processes make the same proposal, and thus the command is committed at partition x with timestamp 6. Similarly, at partition y, `F` computes $\mathbf{10}$ as its proposal and sends it to $\{\texttt{F}, \texttt{G}, \texttt{H}\}$, all of which propose the same. The command is thus committed at partition y with timestamp 10. The final timestamp of the command is then computed as $\max\{6, 10\} = 10$.

[3]When a command accesses a single partition, these are messages from a process to itself, and thus, are processed immediately.

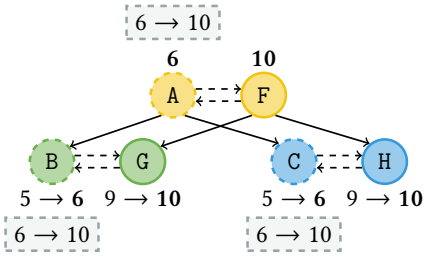


Figure 4.4: Example of Tempo with 2 partitions. Next to each process we show the clock updates upon receiving MPropose messages and, in dashed boxes, the updates upon receiving MCommit or MBump messages (whichever occurs first).

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 28). When A receives these promises, it declares timestamp 10 stable. This occurs after two extra message delays: an MCommit from A and F to B and C, and then MPromises from B and C back to A. Since the command's timestamp is stable at both A and F, once these processes exchange MStable messages, the command can finally be executed at each.

Faster stability

Tempo avoids the above extra delays by generating the detached promises needed for stability earlier than in the MCommit handler. For this, after processing an MPropose message, a process also sends an MBump message containing its proposal to the nearby processes that replicate a partition accessed by the command (line 17). Upon receiving this message (line 18), a process bumps its $\texttt{Clock}_p$ to the timestamp in the message, generating detached promises.

In Figure 4.4, 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 reception, B bumps its $\texttt{Clock}_p$ 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 necessary 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 28) are still necessary for multi-partition commands.

4.2.6 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 (§3). We detail the main differences between the two recovery mechanisms at the end of this section. Tempo recovery protocol at a process $i \in \mathbb{I}_p$ is given in Algorithm 4.3. We use $\texttt{initial}_p(id)$ 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 command with identifier id by calling function $\textbf{recover}(id)$ at line 63. Only a process with $id \in pending$ can take over as a coordinator (line 64): 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 65) 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 $\texttt{bal}[id]$ (line 68). If $\texttt{bal}[id]$ is still 0 (line 69), the process checks the command's phase to decide if it should compute its timestamp proposal for the command. If $id \in payload$ (line 70), the process has not yet computed a timestamp proposal, and thus it does so at line 71. It also sets the command's phase to RECOVER-R, which records that the timestamp proposal was computed in the MRec handler. Otherwise, if $id \in propose$ (line 73), the process has already computed a timestamp proposal at line 15. In this case, the process simply sets the command's phase to RECOVER-P, which records that the timestamp proposal was computed in the MPropose handler. Finally, the process sets $\texttt{bal}[id]$ to the new ballot and replies with an MRecAck message containing the timestamp (ts), the command's phase (phase) and the ballot at which the timestamp was previously accepted in consensus (abal). Note that $\texttt{abal}[id] = 0$ if the process has not yet accepted any consensus proposal. Also note that lines 70 and 73 are exhaustive: these are the only possible phases when $id \in pending$ (line 68) and $\texttt{bal}[id] = 0$ (line 69), as the remaining phases RECOVER-P and RECOVER-R (recall that $pending = payload \cup propose \cup recoverp \cup recoverr$) have non-zero ballots due to line 75.

In the MRecAck handler (line 77), the new coordinator computes the command'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 coordinator 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 about this timestamp. To maintain Invariant 4.1, if any process previously accepted a consensus proposal (line 79), by the standard Paxos rules [7, 35], the coordinator selects the proposal accepted at the highest ballot (line 80).

If no consensus proposal has been accepted before, the new coordinator first computes at line 83 the set of processes I that belong both to the recovery quorum Q and the fast quorum $\texttt{quorums}[id][p]$. Then, depending on whether the initial coordinator replied and in which handler the processes in I have

Algorithm 4.3: Tempo recovery protocol at process $i \in \mathbb{I}_p$.

```
63 recover(id)
64   pre: id ∈ pending
65   b ← i + r(⌊(bal[id]−1)/r⌋ + 1)
66   send MRec(id, b) to 𝕀_p
67 receive MRec(id, b) from j
68   pre: id ∈ pending ∧ bal[id] < b
69   if bal[id] = 0 then
70     if id ∈ payload then
71       ts[id] ← proposal(id, 0)
72       phase[id] ← RECOVER-R
73     else if id ∈ propose then
74       phase[id] ← RECOVER-P
75   bal[id] ← b
76   send MRecAck(id, ts[id], phase[id], abal[id], b) to j
77 receive MRecAck(id, t_j, ph_j, ab_j, b) from ∀j ∈ Q
78   pre: bal[id] = b ∧ |Q| = r − f
79   if ∃k ∈ Q · ab_k ≠ 0 then
80     let k be such that ab_k is maximal
81     send MConsensus(id, t_k, b) to 𝕀_p
82   else
83     I ← Q ∩ quorums[id][p]
84     s ← initial_p(id) ∈ I ∨ ∃k ∈ I · ph_k = RECOVER-R
85     Q' ← if s then Q else I
86     t ← max{t_j | j ∈ Q'}
87     send MConsensus(id, t, b) to 𝕀_p
```

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 ($s = true$, line 84).* In either of these two cases the initial coordinator could not have taken the fast path. If the initial coordinator replies ($\texttt{initial}_p(id) \in I$), then it has not taken the fast path before receiving the MRec message from the new one, as it would have $id \in commit \cup execute$ and the MRec precondition requires $id \in pending$ (line 68). It will also not take the fast path in the future, since when processing the MRec message it sets the command's phase to RECOVER-P (line 74), which invalidates the MProposeAck precondition (line 22). 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 RECOVER-R (line 72), 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 4.1. Since $|Q| = r - f \geq r - \lfloor \frac{r-1}{2} \rfloor \geq \lfloor \frac{r}{2} \rfloor + 1$, the new coordinator has the output of **proposal**

by a majority of processes, and thus it computes the command's timestamp with max (line 86), respecting Property 4.1.

2) The initial coordinator does not reply and all processes in I have computed their timestamp proposal in the MPropose *handler ($s = \mathit{false}$, line 84).* In this case the initial coordinator could have taken the fast path with some timestamp $t = \max\{t_j \mid j \in \texttt{quorums}[id][p]\}$ 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 $\texttt{quorums}[id][p]$ has size $\lfloor \frac{r}{2} \rfloor + f$, the set of processes $I = Q \cap \texttt{quorums}[id][p]$ contains at least $\lfloor \frac{r}{2} \rfloor$ 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 RECOVER-P (line 74), 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 4.2, the new coordinator recovers t by selecting the highest timestamp reported in I (line 86).

4.2.7 Liveness Protocol

Tempo liveness protocol in Algorithm 4.4 builds on the Paxos liveness protocol presented in §2.3. Tempo uses Ω, the leader election failure detector [28], which ensures that from some point on, all correct processes nominate the same correct process as the leader (§2.1). Tempo runs an instance of Ω per partition p. In Algorithm 4.4, $\texttt{leader}_p$ denotes the current leader nominated for partition p at process i. We say that $\texttt{leader}_p$ stabilizes when it stops changing at all correct processes in p. Tempo also uses $\mathbb{I}_c^i$, which we call the *partition covering* failure detector (§4.2.5). At each process i and for every command c, $\mathbb{I}_c^i$ returns a set of processes J such that, for every partition p accessed by c, J contains one process close to i that replicates p. Eventually, $\mathbb{I}_c^i$ only returns correct processes. In the definition above, the closeness between replicas is measured in terms of latency. Returning close replicas is needed for performance but not necessary for the liveness of the protocol. Both $\mathbb{I}_c^i$ and Ω are easily implementable under our assumption of eventual synchrony (§4.1).

For every command id with $id \in \mathit{pending}$ (line 88), process i is allowed to invoke $\mathbf{recover}(id)$ at line 91 only if it is the leader of partition p according to $\texttt{leader}_p$. Furthermore, it only invokes $\mathbf{recover}(id)$ at line 91 if it has not yet participated in consensus (i.e., $\texttt{bal}[id] = 0$) or, if it did, the consensus was lead by another process (i.e., $\mathbf{bal\_leader}(\texttt{bal}[id]) \neq i$). In particular, process i does not invoke $\mathbf{recover}(id)$ at line 91 if it is the leader of $\texttt{bal}[id]$ (i.e., if $\mathbf{bal\_leader}(\texttt{bal}[id]) = i$). This ensures that process i does disrupt a recovery lead by itself.

For a leader to make progress with some $\texttt{MRec}(id, b)$ message, it is required that $r - f$ processes (line 78) have their $\texttt{bal}[id] < b$ (line 68). This may not always be the case because before the variable $\texttt{leader}_p$ stabilizes, any process can invoke $\mathbf{recover}(id)$ at line 91 if it thinks it is the leader. To help the leader select a high enough ballot, and thus ensure it will make progress, we introduce a new message type, MRecNAck. A process sends an $\texttt{MRecNAck}(id, \texttt{bal}[id])$ at line 95 when it receives

Algorithm 4.4: Tempo liveness protocol at process $i \in \mathbb{I}_p$.

```
88  periodically
89    for id ∈ pending
90      send MPayload(id, cmd[id], quorums[id]) to 𝕀_{cmd[id]}
91      if leader_p = i ∧ (bal[id] = 0 ∨ bal_leader(bal[id]) ≠ i) then
92        recover(id)
93  receive MConsensus(id, _, b) or MRec(id, b) from j
94    pre: bal[id] > b
95    send MRecNAck(id, bal[id]) to j
96  receive MRecNAck(id, b)
97    pre: leader_p = i ∧ bal[id] < b
98    bal[id] ← b
99    recover(id)
100 receive MCommitRequest(id) from j
101   pre: id ∈ commit ∪ execute
102   send MPayload(id, cmd[id], quorums[id]) to j
103   send MCommit(id, ts[id]) to j
104 bal_leader(b)
105   return b − r ∗ ⌊(b−1)/r⌋
```

an $\texttt{MConsensus}(id, \_, b)$ or $\texttt{MRec}(id, b)$ (line 93) with some ballot number b lower than its $\texttt{bal}[id]$ (line 94). When process i receives an $\texttt{MRecNAck}(id, b)$ with some ballot number b higher than its $\texttt{bal}[id]$, if it is still the leader (line 97), it joins ballot b (line 98) and invokes $\textbf{recover}(id)$ again. This results in process i sending a new MRec with some ballot higher than b lead by itself. As only $\texttt{leader}_p$ is allowed to invoke $\textbf{recover}(id)$ at line 91 and line 97, and since $\texttt{leader}_p$ eventually stabilizes, this mechanism ensures that eventually the stable leader will start a high enough ballot in which enough processes will participate.

Given a command c with identifier id submitted by a correct process, a correct process in $\mathbb{I}_c$ can only commit c if it has $id \in pending$ and receives an $\texttt{MCommit}(id, \_)$ from every partition accessed by c (line 26). Next, we let p be one such partition, and we explain how every correct process in $\mathbb{I}_c$ eventually has $id \in pending$ and receives such an $\texttt{MCommit}(id, \_)$ sent by some correct process in $\mathbb{I}_p$. We consider two distinct scenarios.

In the first scenario, some correct process $i \in \mathbb{I}_p$ already has $id \in commit \cup execute$. This scenario is addressed by adding two new lines to the MPromises handler: line 51 and line 52. Since $id \in commit \cup execute$ at i, by Property 4.1 there is a majority of processes in each partition q accessed by c that have called $\textbf{proposal}(id, \_)$, and have thus generated a promise attached to id. Moreover, given that at most f processes can fail, at least one process from such a majority is correct. Let one of these processes be $j \in \mathbb{I}_q$. Due to line 47, process j periodically sends an MPromises message that contains a promise attached to id. When a process $k \in \mathbb{I}_q$ receives such a message, if id is neither committed nor executed locally (line 51), k sends an $\texttt{MCommitRequest}(id)$ to $\mathbb{I}_c$. In particular, it

sends such message to process $i \in \mathbb{I}_p$. As $id \in commit \cup execute$ at process i, when i receives the $\texttt{MCommitRequest}(id)$ (line 100), it replies with an $\texttt{MPayload}(id, \_, \_)$ and an $\texttt{MCommit}(id, \_)$. In this way, any process $k \in \mathbb{I}_q$ will eventually have $id \in pending$ and receive an $\texttt{MCommit}(id, \_)$ sent by some correct process in $\mathbb{I}_p$, as required.

In the second scenario, no correct process in $\mathbb{I}_p$ has $id \in commit \cup execute$ but some correct process in $\mathbb{I}_c$ has $id \in pending$. Due to line 90, such process sends an $\texttt{MPayload}$ message to $\mathbb{I}_c$. Thus, every correct process in $\mathbb{I}_p$ eventually has $id \in pending$. In particular, this allows $\texttt{leader}_p$ to invoke $\textbf{recover}(id)$ and the processes in $\mathbb{I}_p$ to react to the $\texttt{MRec}(id, \_)$ message by $\texttt{leader}_p$. Hence, after picking a high enough ballot using the mechanism described earlier, $\texttt{leader}_p$ will eventually send an $\texttt{MCommit}(id, \_)$ to $\mathbb{I}_c$, as required.

Finally, note that calling $\textbf{recover}(id)$ for any command id such that $id \in pending$ (line 88) can disrupt the fast path. This does not affect liveness but may degrade performance. Thus, to attain good performance, any implementation of Tempo should only start calling $\textbf{recover}(id)$ after some reasonable timeout on id. In order to save bandwidth, the sending of MCommitRequest messages can also be delayed in the hope that such information will be received anyway.

Atlas recovery vs Tempo recovery As described in §3.2.2, there are situations in Atlas where the payload of some command id is lost and some other command id' depends on it (i.e., $id \in \texttt{dep}[id']$). In this case, without the information about id, the execution mechanism of Atlas will simply block, and thus the protocol has to allow id to be committed even though its payload is lost. Since Atlas may have to commit commands with unknown payloads, the protocol introduces a special noOp command. As a result, a command can be committed with the payload initially supplied by the client or noOp when such payload is lost. It is due to this that Atlas includes the command payload into MConsensus messages, thus ensuring that a unique payload will be chosen (Invariant 3.1). Tempo does not require a special noOp command, and thus, MConsensus messages do not have to include the command payload. The noOp command is avoided by Tempo because its execution mechanism can never block (as long as at most f processes have failed). We prove this formally in §4.3.

The second difference between the two recovery mechanisms is in the preconditions of $\textbf{recover}(id)$ and MRec. As we just mentioned, Atlas has to allow any command to be recovered, and thus $\textbf{recover}(id)$ has no precondition. Tempo, however, requires that $id \in pending$ (lines 64 and 68), which ensures that the payload of the command being recovered is known by the process performing recovery.

Finally, the two protocols track whether a fast-quorum process has seen the initial message by the coordinator (MCollect in Atlas and MPropose in Tempo) in different ways. In Atlas a fast-quorum process has seen the initial message when the fast quorum is known (line 50 in Algorithm 3.2), and in Tempo when the command phase at the process is RECOVER-P (line 84 in Algorithm 4.3). This is mostly a cosmetic difference and Atlas could have also used phases to track this information.

4.2.8 Properties

In §3.2.4 we have analyzed two properties of the Atlas protocol: its complexity and fault-tolerance. Such analysis also applies to the Tempo protocol. We now discuss two other properties of Tempo.

Genuineness and parallelism

The above protocol is *genuine*: for every command c, only the processes in $\mathbb{I}_c$ take steps to order and execute c [39]. This is not the case for existing leaderless protocols for partial replication, such as Janus [15]. With a genuine protocol, partitioning the application state brings scalability in parallel workloads: an increase in the number of partitions (and thereby of available machines) leads to an increase in throughput. When partitions are colocated in the same machine, the message passing in Algorithm 4.1 and Algorithm 4.2 can be optimized and replaced by shared-memory operations. Since Tempo runs an independent instance of the protocol for each partition replicated at the process, the resulting protocol is highly parallel.

4.2.9 Optimizations

In §3.2.5, we have introduced two techniques that can improve the performance of Atlas: *non-fault tolerant reads* (NFR) and *avoiding a fast-quorum straggler*. Tempo can also be improved using similar strategies. In this section we describe how the first one is implemented in Tempo.

Non-fault-tolerant reads. Similarly to Atlas, we observe that single-partition reads can be handled in a more efficient way. In Algorithm 4.1, when proposing a timestamp for a command, a process always bumps $\texttt{Clock}_p$ (line 37) and generates an attached vote with its proposal (line 39). With NFR, single-partition reads do not have attached votes and only bump clocks if detached votes are needed for timestamp stability. For this, we change the $\texttt{Clock}_p + 1$ in lines 6 and 37 to be simply $\texttt{Clock}_p$ and eliminate line 39 where attached votes are generated. Because a read never has votes attached to it, it will never block a later command, even if it is not fully executed, e.g., when its coordinator fails (or hangs). For this reason, and like Atlas, single-partition reads can be executed in a non-fault-tolerant manner. More precisely, for some read with identifier id, the coordinator selects a plain majority as a fast quorum (line 3), independently of the value of f. Then, at the end of the PROPOSE phase, it immediately commits id, setting $\texttt{ts}[id]$ to the maximum of all timestamp proposals returned by this quorum (line 23).

4.3 Correctness

In this section we prove that the Tempo protocol satisfies the PSMR specification (§4.1). We omit the trivial proof of Validity, and first prove Ordering and then Liveness.

Consider the auxiliary invariants below:

Invariant 4.3. Assume $\texttt{MConsensus}(\_, \_, b)$ has been sent by process i. Then $b = i$ or $b > r$.

Invariant 4.4. Assume $\texttt{MConsensus}(id, t, b)$ and $\texttt{MConsensus}(id, t', b')$ have been sent. If $b = b'$, then $t = t'$.

Invariant 4.5. Assume $\texttt{MRecAck}(\_, \_, \_, ab, b)$ has been sent by some process. Then $ab < b$.

Invariant 4.6. Assume $\texttt{MConsensusAck}(id, b)$ and $\texttt{MRecAck}(id, \_, \_, ab, b')$ have been sent by some process. If $b' > b$, then $b \leq ab < b'$ and $ab \neq 0$.

Invariant 4.7. If $id \notin start$ at some process then the process knows $\texttt{cmd}[id]$ and $\texttt{quorums}[id]$.

Invariant 4.8. If a process executes the $\texttt{MConsensusAck}(id, \_)$ or $\texttt{MRecAck}(id, \_, \_, \_, \_)$ handlers then $id \notin start$.

Invariant 4.9. Assume a slow quorum has received $\texttt{MConsensus}(id, t, b)$ and responded to it with $\texttt{MConsensusAck}(id, b)$. For any $\texttt{MConsensus}(id, t', b')$ sent, if $b' > b$, then $t' = t$.

Invariant 4.10. Assume $\texttt{MCommit}(id, t)$ has been sent at line 24. Then for any $\texttt{MConsensus}(id, t', \_)$ sent, $t' = t$.

Invariants 4.3-4.8 easily follow from the structure of the protocol. Next we prove the rest of the invariants. Then we prove Invariant 4.1 and Property 4.1 (the latter is used in the proof of Theorem 4.1). Finally, we introduce and prove two lemmas that are then used to prove the Ordering property of the PSMR specification.

Proof of Invariant 4.9. Assume that at some point

> (*) a slow quorum has received $\texttt{MConsensus}(id, t, b)$ and responded to it with $\texttt{MConsensusAck}(id, b)$.

We prove by induction on b' that, if a process i sends $\texttt{MConsensus}(id, t', b')$ with $b' > b$, then $t' = t$. Given some b^*, assume this property holds for all $b' < b^*$. We now show that it holds for $b' = b^*$. We make a case split depending on the transition of process i that sends the $\texttt{MConsensus}$ message.

First, assume that process i sends $\texttt{MConsensus}$ at line 25. In this case, $b' = i$. Since $b' > b$, we have $b < i$. But this contradicts Invariant 4.3. Hence, this case is impossible.

The remaining case is when process i sends $\texttt{MConsensus}$ during the transition at line 77. In this case, i has received

$$\texttt{MRecAck}(id, t_j, \_, ab_j, b')$$

from all processes j in a recovery quorum Q^R. Let $ab_{\max} = \max\{ab_j \mid j \in Q^R\}$; then by Invariant 4.5 we have $ab_{\max} < b'$.

Since the recovery quorum Q^R has size $r - f$ and the slow quorum from (*) has size $f + 1$, we get that at least one process in Q^R must have received the $\texttt{MConsensus}(id, t, b)$ message and responded to it with $\texttt{MConsensusAck}(id, b)$. Let one of these processes be l. Since $b' > b$, by Invariant 4.6 we have $ab_l \neq 0$, and thus process i executes line 81. By Invariant 4.6 we also have $b \leq ab_l$ and thus $b \leq ab_{\texttt{max}}$.

Consider an arbitrary process $k \in Q^R$, selected at line 80, such that $ab_k = ab_{\texttt{max}}$. We now prove that $t_k = t$. If $ab_{\texttt{max}} > b$, then since $ab_{\texttt{max}} < b'$, by induction hypothesis we have $t_k = t$, as required. If $ab_{\texttt{max}} = b$, then since $ab_{\texttt{max}} \neq 0$, process k has received some $\texttt{MConsensus}(id, \_, ab_{\texttt{max}})$ message. By Invariant 4.4, process k must have received the same $\texttt{MConsensus}(id, t, ab_{\texttt{max}})$ received by process l. Upon receiving this message, process k stores t in $\texttt{ts}$ and does not change this value at line 71: $ab_{\texttt{max}} \neq 0$ and thus $\texttt{bal}[id]$ cannot be 0 at line 69. Thus process k must have sent $\texttt{MRecAck}(id, t_k, \_, ab_{\texttt{max}}, b')$ with $t_k = t$, which concludes the proof. □

Proof of Invariant 4.10. Assume $\texttt{MCommit}(id, t)$ has been sent at line 24. Then the process that sent this $\texttt{MCommit}$ message must be process $\texttt{initial}_p(id)$. Moreover, we have that for some fast quorum mapping Q^F such that $\texttt{initial}_p(id) \in Q^F[p]$:

> (*) every process $j \in Q^F[p]$ has received $\texttt{MPropose}(id, c, Q^F, m)$ and responded with $\texttt{MProposeAck}(id, t_j)$ such that $t = \max\{t_j \mid j \in Q^F[p]\}$.

We prove by induction on b that, if a process i sends $\texttt{MConsensus}(id, t', b)$, then $t' = t$. Given some b^*, assume this property holds for all $b < b^*$. We now show that it holds for $b = b^*$.

First note that process i cannot send $\texttt{MConsensus}$ at line 25, since in this case we would have $i = \texttt{initial}_p(id)$, and $\texttt{initial}_p(id)$ took the fast path at line 24. Hence, process i must have sent $\texttt{MConsensus}$ during the transition at line 77. In this case, i has received

$$\texttt{MRecAck}(id, t_j, ph_j, ab_j, b)$$

from all processes j in a recovery quorum Q^R.

If $\texttt{MConsensus}$ is sent at line 81, then we have $ab_k > 0$ for the process $k \in Q^R$ selected at line 80. In this case, before sending $\texttt{MRecAck}$, process k must have received

$$\texttt{MConsensus}(id, t_k, ab_k)$$

with $ab_k < b$. Then by induction hypothesis we have $t' = t_k = t$. This establishes the required.

If $\texttt{MConsensus}$ is not sent in line 81, then we have $ab_k = 0$ for all processes $k \in Q^R$. In this case, process i sends $\texttt{MConsensus}$ in line 87. Since the recovery quorum Q^R has size $r - f$ and the fast quorum $Q^F[p]$ from (*) has size $\lfloor \frac{r}{2} \rfloor + f$, we have that

> (**) at least $\lfloor \frac{r}{2} \rfloor$ processes in Q^R are part of $Q^F[p]$ and thus must have received $\texttt{MPropose}(id, c, Q^F, m)$ and responded to it with $\texttt{MProposeAck}$.

Let I be the set of processes $Q^R \cap Q^F[p]$ (line 83). By our assumption, process $\texttt{initial}_p(id)$ sent an $\texttt{MCommit}(id, t)$ at line 24, and thus $id \in commit \cup execute$ at this process. Then due to the check $id \in pending$ at line 68, this process did not reply to MRec. Hence, $\texttt{initial}_p(id)$ is not part of the recovery quorum, i.e., $\texttt{initial}_p(id) \notin I$ at line 84. Moreover, since the initial coordinator takes the fast path at line 24, all fast quorum processes have set $\texttt{phase}[id]$ to PROPOSE when processing the MPropose from the coordinator (line 14). Due to this and to the check at line 73, their $\texttt{phase}[id]$ value is set to RECOVER-P in line 74, and thus, we have that all fast quorum processes that replied report RECOVER-P in their MRecAck message, i.e., $\forall k \in I \cdot ph_k = \texttt{RECOVER-P}$ at line 84. It follows that the condition at line 84 does not hold and thus the quorum selected is I (line 85). By Property 4.2, the fast-path proposal $t = \max\{t_j \mid j \in Q^F[p]\}$ can be obtained by selecting the highest proposal sent in MPropose by any $\lfloor \frac{r}{2} \rfloor$ fast-quorum processes (excluding the initial coordinator). By (**), and since all processes $k \in I$ have $ab_k = 0$, then all processes in I replied with the timestamp proposal that was sent to the initial coordinator. Thus, by Property 4.2 we have $t = \max\{t_j \mid j \in Q^F[p]\} = \max\{t_j \mid j \in I\} = t'$, which concludes the proof. □

Proof of Invariant 4.1. Assume that $\texttt{MCommit}(id, t)$ and $\texttt{MCommit}(id, t')$ have been sent. We prove that $t = t'$.

Note that, if an $\texttt{MCommit}(id, t)$ was sent at line 103, then some process sent an $\texttt{MCommit}(id, t)$ at line 24 or line 35. Hence, without loss of generality, we can assume that the two MCommit under consideration were sent at line 24 or at line 35. We can also assume that the two MCommit have been sent by different processes. Only one process can send an MCommit at line 24 and only once. Hence, it is sufficient to only consider the following two cases.

Assume first that both MCommit messages are sent at line 35. Then for some b, a slow quorum has received $\texttt{MConsensus}(id, t, b)$ and responded to it with $\texttt{MConsensusAck}(id, b)$. Likewise, for some b', a slow quorum has received $\texttt{MConsensus}(id, t', b')$ and responded to it with $\texttt{MConsensusAck}(id, b')$. Assume without loss of generality that $b \leq b'$. If $b < b'$, then $t' = t$ by Invariant 4.9. If $b = b'$, then $t' = t$ by Invariant 4.4. Hence, in this case $t' = t$, as required.

Assume now that $\texttt{MCommit}(id, t)$ was sent at line 24 and $\texttt{MCommit}(id, t')$ at line 35. Then for some b, a slow quorum has received $\texttt{MConsensus}(id, t', b)$ and responded to it with $\texttt{MConsensusAck}(id, b)$. Then by Invariant 4.10, we must have $t' = t$, as required. □

Proof of Property 4.1. Assume that $\texttt{MCommit}(id, t)$ has been sent. We prove that there exists a quorum $\widehat{Q}$ with $|\widehat{Q}| \geq \lfloor \frac{r}{2} \rfloor + 1$ and $\widehat{t}$ such that $t = \max\{\widehat{t}_j \mid j \in \widehat{Q}\}$, where each process $j \in \widehat{Q}$ computes its $\widehat{t}_j$ in either line 15 or line 71 using function **proposal** and sends it in either $\texttt{MProposeAck}(id, \widehat{t}_j)$ or $\texttt{MRecAck}(id, \widehat{t}_j, \_, 0, \_)$.

The computation of t occurs either in the transition at line 23 or at line 86. If the computation of t occurs at line 23, then the quorum Q defined at line 22 is a fast quorum with size $\lfloor \frac{r}{2} \rfloor + f$. In this case, we let $\widehat{Q} = Q$ and $\forall j \in Q \cdot \widehat{t}_j = t_j$, where t_j is defined at line 21. Since $|Q| = \lfloor \frac{r}{2} \rfloor + f$ and $f \geq 1$,

we have $|\widehat{Q}| \geq \lfloor \frac{r}{2} \rfloor + 1$, as required. If the computation of t occurs at line 86, we have two situations depending on the condition at line 84. Let I be the set computed at line 83, i.e., the intersection between the recovery quorum Q (defined at line 77) and the fast quorum $\texttt{quorums}[id][p]$ ($\texttt{quorums}[id][p]$ is known by Invariant 4.7 and Invariant 4.8). First, consider the case in which the condition at line 84 holds. In this case, we let $\widehat{Q} = Q$ and $\forall j \in Q \cdot \widehat{t}_j = t_j$, where t_j is defined at line 77. Since $|Q| = r - f$ and $f \leq \lfloor \frac{r-1}{2} \rfloor$, we have $|\widehat{Q}| \geq \lfloor \frac{r}{2} \rfloor + 1$, as required. Now consider the case in which the condition at line 84 does not hold. Given that the fast quorum size is $\lfloor \frac{r}{2} \rfloor + f$ and the size of the recovery quorum Q is $r - f$, we have that I contains at least $(\lfloor \frac{r}{2} \rfloor + f) - f = \lfloor \frac{r}{2} \rfloor$ fast quorum processes. Note that, since t is computed at line 86, we have that $\forall k \in Q \cdot ab_k = 0$ (line 79). For this reason, each process in Q had $\texttt{bal}[id] = 0$ (line 69) when it received the first $\texttt{MRec}(id, \_)$ message. Moreover, as each process in I reported RECOVER-P (i.e., $\forall k \in I \cdot ph_k = \texttt{RECOVER-P}$ at line 84), the $\texttt{phase}[id]$ was PROPOSE (line 73) when the process received the first $\texttt{MRec}(id, \_)$ message. It follows that each of these processes computed their timestamp proposal in the MPropose handler at line 15 (not in the MRec handler at line 71). Thus, these processes have proposed a timestamp at least as high as the one from the initial coordinator. In this case, we let $\widehat{Q} = I \cup \{\texttt{initial}_p(id)\}$, $\forall j \in I \cdot \widehat{t}_j = t_j$ where t_j is defined at line 77 and $\widehat{t}_{\texttt{initial}_p(id)}$ be the timestamp sent by $\texttt{initial}_p(id)$ in its $\texttt{MProposeAck}(id, \_)$ message. Since $|I| \geq \lfloor \frac{r}{2} \rfloor$ and $\texttt{initial}_p(id) \notin I$, we have $|\widehat{Q}| \geq \lfloor \frac{r}{2} \rfloor + 1$, as required. □

We now prove that Tempo ensures Ordering. First we introduce the following two lemmas. Consider two commands c and c' submitted during a run of Tempo with identifiers id and id'. By Invariant 4.1, all the processes agree on the final timestamp of a command. Let t and t' be the final timestamp of c and c', respectively.

Lemma 4.1. If $c \mapsto_i c'$ then $\langle t, id \rangle < \langle t', id' \rangle$.

Proof. By contradiction, assume that $c \mapsto_i c'$, but $\langle t', id' \rangle < \langle t, id \rangle$. Consider the point in time when i executes c (line 60). By Validity, this point in time is unique. Since $c \mapsto_i c'$, process i cannot have executed c' before this time. Process i may only execute c once it is in ids (line 56). Hence, $t \leq h[\lfloor \frac{r}{2} \rfloor]$ (line 55). From Theorem 4.1, t is stable at i. As $\langle t', id' \rangle < \langle t, id \rangle$, we get $t' \leq t$, and by Invariant 4.2, $id' \in commit \cup execute$ at i. Then, since $t' \leq t \leq h[\lfloor \frac{r}{2} \rfloor]$ and c' cannot have been executed before c at process i, we have $id' \in ids$. But then as $\langle t', id' \rangle < \langle t, id \rangle$, c' is executed before c at process i (line 56). This contradicts $c \mapsto_i c'$. □

Lemma 4.2. If $c \mapsto c'$ then whenever a process i executes c', some process has already executed c.

Proof. Assume that a process i executed c'. By the definition of $\mapsto$, either $c \rightsquigarrow c'$, or $c \mapsto_j c'$ at some process j. Assume first that $c \rightsquigarrow c'$. By definition of $\rightsquigarrow$, c returns before c' is submitted. This requires that command c is executed at least at one replica for each of the partition it accesses. Hence, at the time c' is executed at i, command c has already executed elsewhere, as required.

Assume now that $c \mapsto_j c'$ for some process j. Then c and c' access a common partition, say p, and by Lemma 4.1, $\langle t, id\rangle < \langle t', id'\rangle$. Consider the point in time τ when i executes c'. Before this, according to line 58, process i receives an $\texttt{MStable}(id')$ message from some process $k \in \mathbb{I}_p$. Let $\tau' < \tau$ be the moment when k sends this message by executing line 57. According to line 56, id' must belong to ids at time τ'. Hence, $t' \leq h[\lfloor \frac{r}{2} \rfloor]$ (line 55). From Theorem 4.1, t' is stable at k. As $\langle t, id\rangle < \langle t', id'\rangle$, we have $t \leq t'$, and by Invariant 4.2, $id \in commit \cup execute$ holds at time τ' at process k. Since $t \leq t' \leq h[\lfloor \frac{r}{2} \rfloor]$, either k already executed c, or $id \in ids$. In the latter case, because $\langle t, id\rangle < \langle t', id'\rangle$, command c is executed before k sends the $\texttt{MStable}$ message for id'. Hence, in both cases, c is executed no later than $\tau' < \tau$, as required. □

Proof of Ordering. By Validity, cycles of size one are prohibited. By Lemma 4.2, so are cycles of size two or greater. □

We now prove the Liveness property of the PSMR specification (§4.1). For simplicity, we assume that links are reliable, i.e., if a message is sent between two correct processes then it is eventually delivered. In the following, we use $\texttt{dom}(m)$ to denote the domain of mapping m and $\texttt{img}(m)$ to denote its image. We let id be the identifier of some command c, so that $\mathbb{I}_c$ denotes the set of processes replicating the partitions accessed by c.

Lemma 4.3. Assume that $id \in commit \cup execute$ at some process from a partition p. Then $id \in \texttt{dom}(\texttt{Attached})$ and $id \notin start$ at some correct process from each partition q accessed by command c.

Proof. Since $id \in commit \cup execute$ at a correct process from p, an $\texttt{MCommit}(id, \_)$ has been sent by some process from each partition accessed by c (line 26). In particular, it has been sent by some process from partition q. By Property 4.1, there is a majority of processes Q from partition q that called $\textbf{proposal}(id, \_)$, generating a promise attached to id (line 39), and thus, have $id \in \texttt{dom}(\texttt{Attached})$. Since at most f of these processes can fail, at least some process $j \in Q$ is correct. Moreover, since j has generated a promise attached to id, it is impossible to have $id \in start$ at j (see the $\texttt{MPropose}$ and $\texttt{MRec}$ handlers where $\textbf{proposal}(id)$ is called). Thus $id \in \texttt{dom}(\texttt{Attached})$ and $id \notin start$ at j, as required. □

Lemma 4.4. Assume that $id \notin start$ at some correct process i from partition p. Then eventually every correct process j from some partition q accessed by command c has $id \notin start$ and receives an $\texttt{MCommit}(id, \_)$ sent by some process from partition p.

Proof. We consider the case where $id \notin commit \cup execute$ never holds at j (if it does hold, then j has $id \notin start$ and received an $\texttt{MCommit}(id, \_)$ sent by some process from partition p, as required).

First, assume that $id \in commit \cup execute$ eventually holds at process i. By Lemma 4.3, there exists a correct process k from q that has $id \in \texttt{dom}(\texttt{Attached})$. Due to line 47, k continuously sends an `MPromises` message to all the processes from q, including j. Note that, since $id \in \texttt{dom}(\texttt{Attached})$ at k, this `MPromises` message contains a promise attached to id. Once j receives such a message, since it has $id \notin commit \cup execute$ (line 51), it sends an $\texttt{MCommitRequest}(id)$ to $\mathbb{I}_c$ (line 52), and in particular to process i. Since $id \in commit \cup execute$ at i (line 101), process i replies with an $\texttt{MPayload}(id, \_, \_)$ and $\texttt{MCommit}(id, \_)$. Once j processes such messages, it has $id \notin start$ and received an $\texttt{MCommit}(id, \_)$ by process i, a correct process from partition p, as required.

Now, assume that $id \in pending$ holds forever at process i. Consider the moment τ_0 when the variable $\texttt{leader}_p$ stabilizes. Let process l be $\texttt{leader}_p$. Assume that $id \notin commit \cup execute$ at processes i and l forever (the case where it changes to $id \in commit \cup execute$ at process i is covered above; the case for process l can be shown analogously). Due to line 89, process i sends an $\texttt{MPayload}(id, \_, \_)$ message to $\mathbb{I}_c$ (line 90), in particular to l and j. Once l processes this message, it has $id \in pending$ forever (since we have assumed that $id \notin commit \cup execute$ at l forever). Once j processes this message, it has $id \notin start$, as required. We now prove that eventually process l sends an $\texttt{MCommit}(id, \_)$ to j.

First, we show by contradiction that the number of $\texttt{MRec}(id, \_)$ messages sent by l is finite. Assume the converse. After τ_0, due to the check $\texttt{leader}_p = l$ at line 91 and at line 97, only process l sends $\texttt{MRec}(id, \_)$ messages. Since $\texttt{MRec}(id, \_)$ messages by processes other than l are all sent before τ_0, their number is finite. For this same reason, the number of $\texttt{MConsensus}(id, \_, \_)$ messages by processes other than l are also finite. Thus, each correct process joins only a finite number of ballots that are not owned by l. It follows that the number of $\texttt{MRec}(id, \_)$ messages sent by l at line 92 because it joined a ballot owned by other processes (i.e., when $\textbf{bal\_leader}(\texttt{bal}[id]) \neq l$) is finite. (Note that a single $\texttt{MRec}(id, \_)$ can be sent here due to $\texttt{bal}[id] = 0$, as process l sets $\texttt{bal}[id]$ to some non-zero ballot when processing its first $\texttt{MRec}(id, \_)$ message). For an $\texttt{MRec}(id, \_)$ to be sent at line 99, process l has to receive an $\texttt{MRecNAck}(id, b)$ with $\texttt{bal}[id] < b$ (line 97). If $\textbf{bal\_leader}(b) \neq l$, the number of such `MRecNAck` messages is finite as the number of $\texttt{MRec}(id, \_)$ and $\texttt{MConsensus}(id, \_, \_)$ by processes other than l are finite. If $\textbf{bal\_leader}(b) = l$, the $\texttt{MRecNAck}(id, b)$ must be in response to an $\texttt{MRec}(id, b)$ or $\texttt{MConsensus}(id, \_, b)$ by l. Note that when l sends such a message, it sets $\texttt{bal}[id]$ to b. For this reason, process l cannot have $\texttt{bal}[id] < b$, and hence this case is impossible. Thus, there is a point in time $\tau_1 \geq \tau_0$ after which the condition at line 97 does not hold at process l, and consequently, process l stops sending new $\texttt{MRec}(id, \_)$ messages at line 99, which yields a contradiction.

We have established above that $id \in pending$ at process l forever. We now show that process l sends at least one $\texttt{MRec}(id, \_)$ message. Since $id \in pending$ forever, process l executes line 91 for this id infinitely many times. If l does not send at least one $\texttt{MRec}(id, \_)$ at this line it is because $\texttt{bal}[id] > 0$ and $\textbf{bal\_leader}(\texttt{bal}[id]) = l$ forever. If so, we have two cases to consider depending on the value of $\texttt{bal}[id]$. In the first case, $\texttt{bal}[id] = l$ at l forever. In this case, process l took the slow path by sending to $\mathbb{I}_p$ an `MConsensus` message with ballot l (line 25). We now have two sub-cases. If any process sends an `MRecNAck` to process l (line 95), this will make process l send an $\texttt{MRec}(id, \_)$ (line 99), as required.

Otherwise, process l will eventually gather $f+1$ MConsensusAck messages and commit the command. As we have established that $id \in pending$ at process l forever, this sub-case is impossible. In the second case, we eventually have $\texttt{bal}[id] > l$. Since from some point on, $\textbf{bal\_leader}(\texttt{bal}[id]) = l$ at l, this means that this process sends an $\texttt{MRec}(id, \texttt{bal}[id])$ (line 75), as required.

We have now established that process l sends a finite non-zero number of $\texttt{MRec}(id, \_)$ messages. Let b be the highest ballot for which process l sends an $\texttt{MRec}(id, b)$ message. We now prove that l eventually sends an $\texttt{MCommit}(id, \_)$ to $\mathbb{I}_c$, in particular to j. Given that at most f processes can fail, there are enough correct processes to eventually satisfy the preconditions of MRecAck and MConsensus. First we consider the case where the preconditions of MRecAck and MConsensus eventually hold at process l. Since the precondition of MRecAck eventually holds, process l eventually sends an $\texttt{MConsensus}(id, \_, b)$. Since the precondition of MConsensus eventually holds, process l eventually sends an $\texttt{MCommit}(id, \_)$ to j, as required. Consider now the opposite case where the preconditions of MRecAck or MConsensusAck never hold at process l. Since there are enough correct processes to eventually satisfy these preconditions, the fact that they never hold at process l implies that there is some correct process j with $\texttt{bal}[id] > b$ (otherwise j would eventually reply to process l). Thus, the precondition of MRecNAck holds at j (line 94), which causes j to send an $\texttt{MRecNAck}(id, b')$ with $b' = \texttt{bal}[id] > b$ to l. When l receives such a message, it sends a new $\texttt{MRec}(id, b'')$ with some ballot $b'' > b'$ (line 99). It follows that $b'' > b' > b$, which contradicts the fact that b is the highest ballot sent by l, and hence this case is impossible. □

Lemma 4.5. Assume that $id \notin start$ at some correct process i from some partition p accessed by command c. Then eventually $id \in commit \cup execute$ at every correct process $j \in \mathbb{I}_c$.

Proof. For c to be committed at process j, j has to have $id \notin start$ and to receive an $\texttt{MCommit}(id, \_)$ from each of the partitions accessed by c (line 26). We prove that process j eventually receives such a message from each of these partitions. To this end, fix one such partition q. By Lemma 4.4, it is enough to prove that some correct process from partition q eventually has $id \notin start$. By contradiction, assume that all the correct processes from partition q have $id \in start$ forever. We have two scenarios to consider. First, consider the scenario where $p = q$. But this contradicts the fact that $id \notin start$ at process i. Now, consider the scenario where $p \neq q$. In this scenario we consider two sub-cases. In the first case, eventually $id \in commit \cup execute$ at process i. By Lemma 4.3, there is a correct process from each partition accessed by c, in particular from partition q, that has $id \notin start$, which contradicts our assumption. In the second case, $id \in pending$ at process i forever (the case where it changes to $id \in commit \cup execute$ is covered above). Due to line 90, process i periodically sends an MPayload message to the processes in partition q. Once this message is processed, the correct processes in partition q will have $id \in payload$. Since partition q contains at least one correct process, this also contradicts our assumption. □

Definition 4.1. We define the set of proposals issued by some process $i \in \mathbb{I}_p$ as $\texttt{LocalPromises}_i = \texttt{img}(\texttt{Attached}) \cup \texttt{Detached}$.

Lemma 4.6. For each process $i \in \mathbb{I}_p$ we have that $\langle i, t\rangle \in \texttt{LocalPromises}_i \Rightarrow (\forall u \in \{1, \ldots, t\} \cdot \langle i, u\rangle \in \texttt{LocalPromises}_i)$.

Proof. Follows trivially from Algorithm 4.1 and Algorithm 4.2. □

Lemma 4.7. Consider a command c with an identifier id and the final timestamp t, and assume that $id \notin start$ at some correct process in $\mathbb{I}_c$. Then at every correct process in $\mathbb{I}_c$, eventually variable $\texttt{Promises}$ contains all the promises up to t by some set of processes C with $|C| \geq \lfloor \frac{r}{2} \rfloor + 1$.

Proof. By Lemma 4.5, eventually $id \in commit \cup execute$ at all the correct processes in $\mathbb{I}_c$. Consider a point in time τ_0 when this happens and fix a process $i \in \mathbb{I}_c$ from a partition p that has $id \in commit$. Let C be the set of correct processes from partition p, $\texttt{MPromises}(D_j^0, A_j^0)$ be the $\texttt{MPromises}$ message sent by each process $j \in C$ in the next invocation of line 47 after τ_0, and $ids = \bigcup\{\texttt{dom}(A_j^0) \mid j \in C\}$. Due to line 28, by Lemma 4.6 we have that

$$(*)\ \forall j \in C, u \in \{1, \ldots, t\} \cdot \langle j, u\rangle \in \texttt{img}(A_j^0) \cup D_j^0.$$

Note that, for each $id' \in ids$, we have $id' \notin start$ at some correct process (in particular, at the processes j that sent such id' in their $\texttt{MPromises}(D_j^0, A_j^0)$ messages). Thus, by Lemma 4.5, there exists $\tau_1 \geq \tau_0$ at which $ids \subseteq commit \cup execute$ at process i. Let $\texttt{MPromises}(D_j^1, A_j^1)$ be the $\texttt{MPromises}$ message sent by each process $j \in C$ in the next invocation of line 47 after τ_1. Since at process i we have that $ids \subseteq commit \cup execute$, once all these $\texttt{MPromises}$ messages are processed by process i, for each $j \in C$ we have that $\bigcup\{A_j^1[id'] \mid id' \in ids\} \subseteq \texttt{Promises}$ and $D_j^1 \subseteq \texttt{Promises}$ at process i. Given the definition of ids and since $A_j^0 \subseteq A_j^1$ and $D_j^0 \subseteq D_j^1$, we also have that $\texttt{img}(A_j^0) \subseteq \texttt{Promises}$ and $D_j^0 \subseteq \texttt{Promises}$ at process i. From (*), it follows that $\forall j \in C, u \in \{1, \ldots, t\} \cdot \langle j, u\rangle \in \texttt{Promises}$ at process i. □

Lemma 4.8. Consider a command c with an identifier id, and assume that $id \notin start$ at some correct process in $\mathbb{I}_c$. Then every correct process $i \in \mathbb{I}_c$ eventually executes c.

Proof. Consider a command c with an identifier id, and assume that $id \notin start$ at some correct process in $\mathbb{I}_c$. By contradiction, assume further that some correct process $i \in \mathbb{I}_c$ never executes c. Let $c_0 = c$, $id_0 = id$, and t_0 be the timestamp assigned to c. Then by Lemma 4.7, eventually in every invocation of the periodic handler at line 53, we have $h \leq t_0$. By Lemma 4.5 and since i never executes c_0, eventually i has $id_0 \in commit$. Hence, either i never executes another command c_1 preceding c_0 in ids, or i never receives an $\texttt{MStable}(id_0)$ message from some correct process j eventually indicated

by $\mathbb{I}^{i}_{c_0}$. The latter case can only be due to the loop in the handler at line 53 being stuck at an earlier command c_1 preceding c_0 in ids at j. Hence, in both cases there exists a command c_1 with an identifier id_1 and a final timestamp t_1 such that $\langle t_1, id_1\rangle < \langle t_0, id_0\rangle$ and some correct process $i_1 \in \mathbb{I}_{c_1}$ never executes c_1 despite eventually having $id_1 \in commit$. Continuing the above reasoning, we obtain an infinite sequence of commands $c_0, c_1, c_2, \ldots$ with decreasing timestamp-identifier pairs such that each of these commands is never executed by some correct process. But such a sequence cannot exist because the set of timestamp-identifier pairs is well-founded. This contradiction shows the required. □

Proof of Liveness. Assume that some command c with identifier id is submitted by a correct process or executed at some process. We now prove that it is eventually executed at all correct processes in $\mathbb{I}_c$. By Lemma 4.7 and Lemma 4.8, it is enough to prove that eventually some correct process in $\mathbb{I}_c$ has $id \notin start$. First, assume that id is submitted by a correct process i. Due to the precondition at line 2, $i \in \mathbb{I}_c$. Then $\texttt{phase}[id]$ is set to PROPOSE at process i when i sends the initial MPropose message, and thus $id \notin start$ as required. Assume now that id is executed at some (potentially faulty) process. By Lemma 4.3, there exists some correct process in $\mathbb{I}_c$ with $id \notin start$, as required. □

4.4 Summary and 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 [63–67], usually implemented by a leader-based SMR protocol [7, 8]. As reported in prior works, the leader is a potential bottleneck and is unfair with respect to client locations [11, 62, 68, 69]. To sidestep these problems, leaderless protocols order commands in a fully decentralized manner. Early protocols in this category, such as Mencius [20], 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 [11] and its follow-ups (Atlas (§3), BPaxos [62] and Gryff [9]), order commands by agreeing on a graph of dependencies (§4.2.3). Tempo builds on one of these follow-ups, Atlas (§3), which leverages the observation that correlated failures in geo-distributed systems are rare [4] to reduce the quorum size in leaderless SMR. As we demonstrate next in our evaluation (§5), dependency-based leaderless protocols exhibit high tail latency and suffer from bottlenecks due to their expensive execution mechanism.

Timestamping has been used in two previous leaderless SMR protocols. Caesar [10], which we discussed in §4.2.3, suffers from similar problems to EPaxos. Clock-RSM [21] timestamps each newly submitted command with the coordinator's clock, and then records the association at $f+1$ processes using consensus. Stability occurs when all the processes indicate that their clocks have passed the command's timestamp. As a consequence, the protocol cannot transparently mask failures, like Tempo; these have to be handled via reconfiguration. Its performance is also capped by the speed of the slowest replica, similarly to Mencius [20].

Partial replication is a common way of scaling services that do not fit on a single machine. Some partially replicated systems use a central node to manage access to data, made fault-tolerant via standard SMR techniques [70]. Spanner [4] replaces the central node by a distributed protocol that layers two-phase commit on top of Paxos. Granola [23] follows a similar schema using Viewstamped Replication [71]. Other approaches rely on atomic multicast, a primitive ensuring the consistent delivery of messages across arbitrary groups of processes [39, 57]. Atomic multicast can be seen as a special case of PSMR as defined in §4.1.

Janus [15] generalizes EPaxos to the setting of partial replication. Its authors shows that for a large class of applications that require only one-shot transactions, Janus improves upon prior techniques, including MDCC [72], Tapir [73] and 2PC over Paxos [4]. Our experiments in §5 demonstrate that Tempo significantly outperforms Janus due to its use of timestamps instead of explicit dependencies. Unlike Janus, Tempo is also genuine, which translates into better performance.

Summary This chapter presented Tempo, the first leaderless SMR protocol to determine the order of command execution solely based on scalar timestamps. Tempo cleanly separates timestamp assignment from detecting timestamp stability, and such mechanisms easily extend to partial replication. As we show next in our evaluation (§5), Tempo's approach enables the protocol to offer low tail latency and high throughput even under contended workloads, thus addressing the performance issue inherent to the execution mechanism of EPaxos and Atlas.

5 Performance Evaluation

In this chapter we experimentally evaluate `Atlas` and `Tempo`. In §5.1 we assess whether assuming small values of f is acceptable for geo-distribution. In §5.2 we cover the protocols considered in this evaluation and the framework we developed to implement and evaluate them. In §5.3 we detail our sets of testbeds and benchmarks. Finally, we study the performance of `Atlas` and `Tempo` in full replication (§5.4) and partial replication deployments (§5.5).

5.1 Bounds on Failures

In a practical deployment of `Atlas` and `Tempo`, a critical parameter is the number of concurrent site failures f the protocol can tolerate. It has been reported that concurrent site failures are rare in geo-distributed systems [4]. However, the value of f should also account for asynchrony periods during which sites cannot communicate due to link failures: if more than f sites are unreachable in this way, `Atlas` and `Tempo` may block for the duration of the outage. We have thus conducted an experiment on Google Cloud Platform (GCP) to check that assuming small values of f is still appropriate when this is taken into account.

Our experiment ran for 3 months (October 2018 – January 2019) among 17 sites, the maximal number of sites available in GCP at the time. During the experiment, sites ping each other every second (in the spirit of [17] but on a much larger scale). A link failure occurs between two sites when one of them does not receive a reply after a (tunable) amount of time. Figure 5.1 reports the number of simultaneous link failures for various timeout thresholds. Note that no actual machine crash occurred during the campaign of measurements.

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

From the data collected, we compute the value of f as the smallest number of sites k such that,

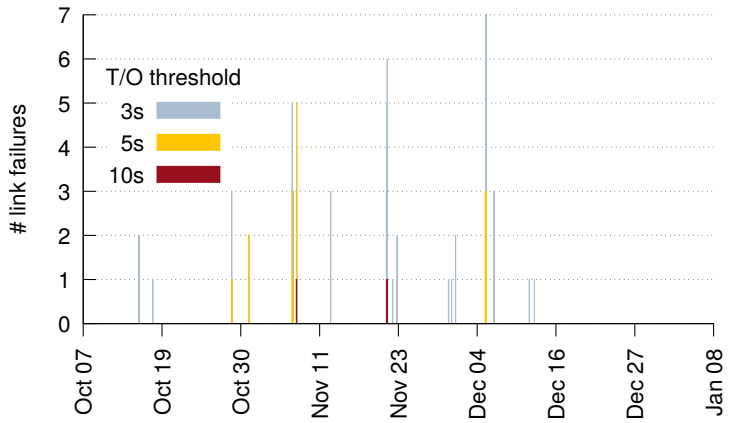


Figure 5.1: The number of simultaneous link failures among 17 sites in GCP when varying the timeout threshold.

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

5.2 Protocols and Implementation

We compare `Atlas` and `Tempo` with Flexible Paxos (FPaxos) [35], EPaxos [11], Caesar [10] and Janus [15]. All these protocols have been briefly covered in §2. FPaxos is a variant of Paxos that, like `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 (§4.2.3). EPaxos and Caesar use fast quorums of size $\lfloor \frac{3r}{4} \rfloor$ and $\lceil \frac{3r}{4} \rceil$, respectively. `Atlas` and `Tempo` use fast quorums of size $\lfloor \frac{r}{2} \rfloor + f$. `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 some of our plots with $r = 5$ 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. 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 [72], Tapir [73] and 2PC over Paxos [4]). Janus 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*.

To improve the fairness of our comparison, all protocols are implemented in the same framework which consists of 35K 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 [74]). 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. Our simulator disregards CPU and network bottlenecks: messages between processes are not sent through sockets, and the time it takes to process each message is not accounted for. 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.

5.2.1 Implementation details

The framework is built using Tokio[1], an asynchronous runtime for Rust with the building blocks needed for writing network applications. Using Tokio, 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[2].

Key-Value Store

We have implemented a simple in-memory key-value store (KVS) that supports two operations: $read(k)$ to fetch the content of the KVS record under key k, and $write(k, v)$ to update its value to v. SMR commands can then access one or more keys at a time[3]. The KVS is used in all our benchmarks (§5.3).

`Atlas` implementation

`Atlas`, EPaxos and Janus rely on sets of dependencies which can grow unboundedly large as the system keeps running. EPaxos [11] solves this issue by compressing a set of dependencies reported by a fast-quorum processes as a vector clock with r entries, where entry j contains the conflicting command with the highest identifier issued by process j. Janus [15] suggests instead that fast-quorum processes report only the direct dependencies of each command (i.e., the most recent conflicting commands seen by each fast-quorum process).

We follow this second approach in the implementation of `Atlas`, EPaxos and Janus. For each key in the KVS, we maintain two command identifiers: the identifier of the latest read-only command (that

[1] https://tokio.rs
[2] fantoch/src/run/task/server/mod.rs
[3] fantoch/src/kvs.rs

accessed such key), and the identifier of the latest write command (where latest means the command most recently seen by the process). In order to allow multiple worker threads to compute and update conflicts (when handling `MCollect` and `MRec` messages), this mapping is maintained in a concurrent hash-map, and each value (i.e. the identifier of a command) is protected by a read-write lock[4]. As we detail next, there are cases in which such values are only read, which explains the use of a read-write lock.

When computing the dependencies of a read-only command with identifier id, for each key accessed by the command, we first retrieve the identifier of the latest write, and then set the latest read to be id. When computing the dependencies of a write command with identifier id, for each key accessed by the command, we first retrieve the latest read and latest write, and then set the latest write to be id. With `NFR` (§3.2.5), the latest read is ignored and only the latest write is retrieved. Overall we have that all commands depend on writes, reads never depend on reads, and writes only depend on reads if `NFR` is disabled.

`Tempo` implementation

We assume that each partition in `Tempo` contains a single key. When partitions/keys are colocated in the same machine, the message passing in `Tempo` is optimized and replaced by shared-memory operations (§4.2.8). For that, the implementation of `Tempo` also makes use of a concurrent hash-map that maps each key to a clock. Instead of protecting this clock with a read-write lock, we use atomics[5]. When computing the timestamp proposal of a command, the clock associated with each key accessed by the command is bumped by one and its new value is fetched. The timestamp proposal is then the highest new clock value fetched. After this, a new round of bumps occurs, incrementing these clocks up to the timestamp proposal computed. With `NFR` (§4.2.9), the timestamp proposal of a single-key read command is simply the current clock value associated with such key.

5.3 Experimental Setup

Testbeds

One of the testbeds uses Amazon EC2 with `c5.2xlarge` instances (machines with 8 virtual CPUs and 16GB of RAM). Another testbed is a local cluster where we inject wide-area delays similar to those observed in EC2. This cluster contains machines with 6 physical cores and 32GB of RAM connected by a 10GBit network. The experiments in Figure 5.5 and Figure 5.6 are performed in EC2, and most of the remaining experiments are conducted on the local cluster. We have validated some of the experiments performed on the local cluster using Google Cloud Platform with an Erlang implementation of `Atlas` [24].

[4] fantoch_ps/src/protocol/common/graph/deps/keys/locked.rs
[5] fantoch_ps/src/protocol/common/table/clocks/keys/atomic.rs

Table 5.1: Ping latency (milliseconds) between Amazon EC2 sites.

	N. California	Singapore	Canada	S. Paulo	Hong Kong	N. Virginia	Tokyo	Stockholm	Mumbai	Oregon
Ireland	141	186	72	183	220	70	204	35	118	127
N. California	–	181	78	190	154	59	111	162	231	21
Singapore		–	221	338	34	235	77	195	63	162
Canada			–	123	202	15	154	114	196	65
S. Paulo				–	315	113	267	217	299	180
Hong Kong					–	213	51	230	98	144
N. Virginia						–	152	105	181	72
Tokyo							–	265	136	100
Stockholm								–	131	157
Mumbai									–	223
Oregon										–

Experiments span up to 11 EC2 regions, which we call *sites*: Ireland (`eu-west-1`), Northern California (`us-west-1`), Singapore (`ap-southeast-1`), Canada (`ca-central-1`), São Paulo (`sa-east-1`). Hong Kong (`ap-east-1`), Northern Virginia (`us-east-1`), Tokyo (`ap-northeast-1`), Stockholm (`eu-north-1`), Mumbai (`ap-south-1`) and Oregon (`us-west-2`). Table 5.1 shows the average ping latencies between these sites. When $r = 3$, the first 3 sites (`eu-west-1`, `us-west-1` and `ap-southeast-1`) are used. When $r = 5$, the next 2 sites (`ca-central-1` and `sa-east-1`) are added, and so on, until we reach $r = 11$.

Benchmarks

We first evaluate full replication deployments (§5.4) initially using a microbenchmark where each command carries a key of 8 bytes and (unless specified otherwise) a payload of 100 bytes. We assume that commands *conflict* when they carry the same key. In the case of `Tempo`, each partition contains a single key, and thus, commands conflict when they access the same partition. To measure performance under a conflict rate ρ of commands, a client chooses key 0 with a probability ρ, and some unique key otherwise. We then evaluate `Atlas` and `Tempo` with a geo-replicated key-value store under the YCSB workload [74]. Finally, we evaluate `Tempo` in partial replication deployments (§5.5) using YCSB+T [75], a transactional version of YCSB. Clients are closed-loop and always deployed in separate machines located in the same regions as servers.

5.4 Full Replication Deployment

Fast-path likelihood

We first evaluate the benefits of `Atlas` and `Tempo` flexible fast-path conditions. To this end, Figure 5.2, Figure 5.3 and Figure 5.4 compare their fast-path ratio with that of EPaxos for different conflict rates and values of f. We also plot *worst-case* scenario, i.e., when all conflicting commands take the slow path. Note that we do not consider $f = 1$ as both `Atlas` and `Tempo` always take the fast path in this case (i.e.,

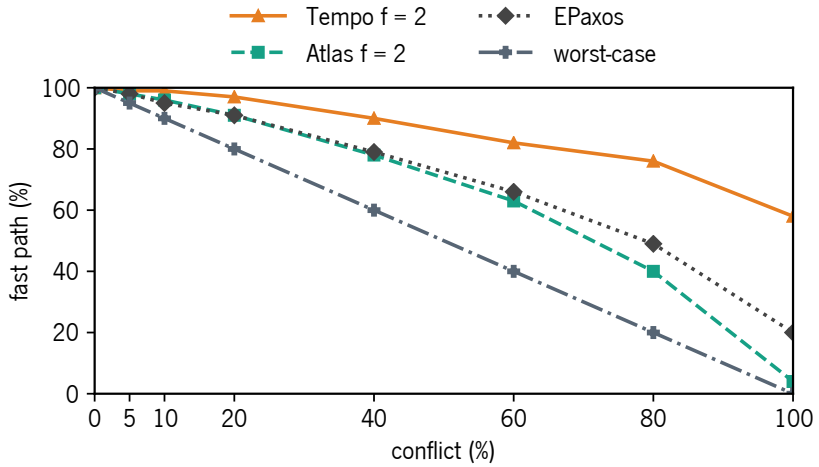


Figure 5.2: Ratio of fast paths for varying conflict rates with $r = 5$ sites and 1 client per site.

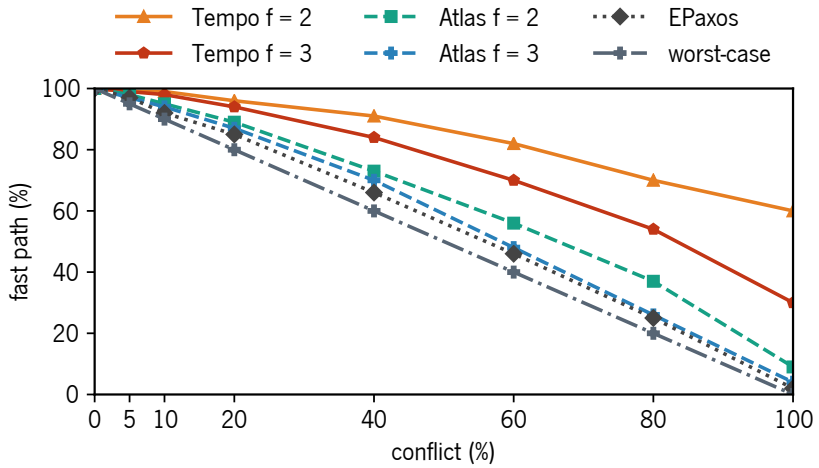


Figure 5.3: Ratio of fast paths for varying conflict rates with $r = 7$ sites and 1 client per site.

the *best-case* scenario). The system consists of $r = 5$ sites in Figure 5.2 and $r = 7$ sites in Figure 5.3, with 1 client per site. In Figure 5.4 we consider $r = 7$ sites but with 8 clients per site instead[6].

In Figure 5.2, with $r = 5$, Atlas $f = 2$ and EPaxos have similar fast-path rates up to 40% conflicts. Even though fast quorum size of Atlas $f = 2$ is larger than EPaxos (4 vs 3), and thus there is a higher chance of fast-quorum processes reporting more conflicts, Atlas flexible fast-path condition compensates for it, allowing the protocol to offer a similar performance to EPaxos. However, this compensation disappears with conflict rates higher than 40%, and EPaxos offers a percentage of fast paths higher than Atlas $f = 2$. In Figure 5.3, the number of sites is increased to $r = 7$. With this setting, EPaxos and Atlas $f = 2$ have exactly the same quorum size (5), and thus performance only depends on the fast-path condition. Due to this, Atlas $f = 2$ is able to provide slightly higher fast-path ratios than EPaxos: these ratios for Atlas $f = 2$ and EPaxos are respectively 89 and 85 (20% conflicts), 73 and 66 (40% conflicts), 56 and 46 (60% conflicts), and 37 and 25 (80% conflicts).

[6]We claim in [24] that the results with 1 or more clients are almost identical. This experiment with more clients shows otherwise.

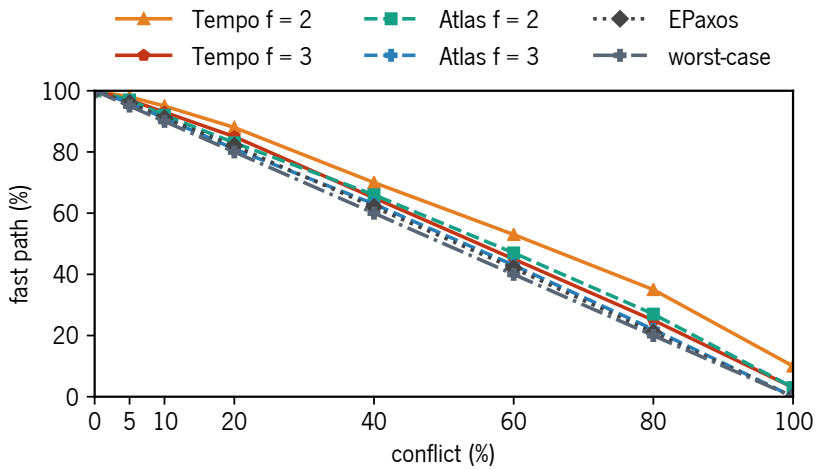


Figure 5.4: Ratio of fast paths for varying conflict rates with $r = 7$ sites and 8 clients per site.

In both Figure 5.2 and Figure 5.3, Tempo $f = 2$ provides noticeable higher fast-path ratios than the other two protocols. For example, with $r = 5$ in Figure 5.2, the ratios for Tempo $f = 2$ and Atlas $f = 2$ are respectively 97 and 91 (20% conflicts), 90 and 78 (40% conflicts), 82 and 63 (60% conflicts), 76 and 40 (80% conflicts), and 58 and 4 (100% conflicts). The difference between Tempo and Atlas may be surprising given that the two protocols have similar flexible fast-path conditions. However, timestamps reported by fast-quorum processes in Tempo simply count the number of conflicts while dependencies in Atlas report exactly which conflicts have occurred. By being less precise in the conflicts detected, Tempo is able to take the fast path more frequently than Atlas[7]. In Figure 5.4, the number of clients is increased from 1 to 8. In this case, the superior performance by Tempo is less apparent since all protocols start approaching the *worst-case* scenario.

Fairness

We now evaluate a key benefit of leaderless SMR, its fairness: the fairer the protocol, the more uniformly it satisfies different sites. We compare Tempo, Atlas, Caesar 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 5.5 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,

[7] To gain intuition on why this is the case, consider the following example with 3 conflicting commands x, y and z. Process A receives first command x and then y, while B receives the commands in the opposite order, first y then x. Now a third command z arrives to these processes. In Atlas, process A reports y as a dependency while B reports x. This disagreement between A and B may force the coordinator of z to take the slow path. This is not the case in Tempo as both A and B will propose the same timestamp for z (say timestamp 3).

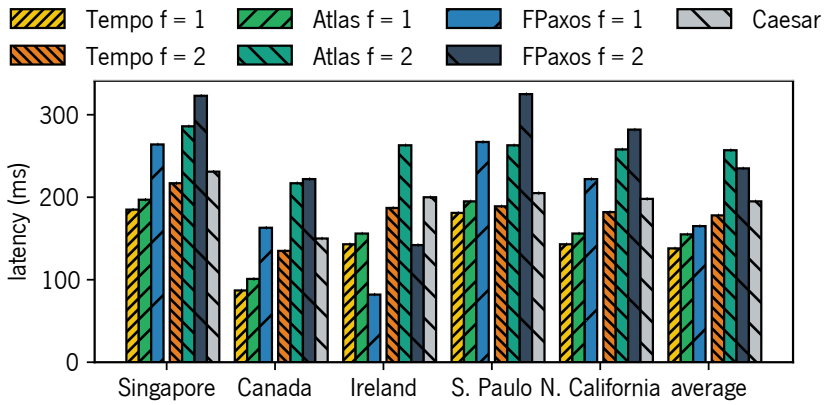


Figure 5.5: Per-site latency with 5 sites and 512 clients per site under a low conflict rate (2%).

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 $\lfloor \frac{r}{2} \rfloor + f$. 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 (§4.2.3). Larger components result in higher average latencies, as reported in Figure 5.5. 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 (§4.2.3), resulting in slightly higher average latencies. As we now demonstrate, both Caesar and `Atlas` have much higher tail latencies than `Tempo`.

Tail latency

Figure 5.6 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 5.5. The protocols are again deployed over 5 EC2 sites.

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 `Atlas` $f = 1$, 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

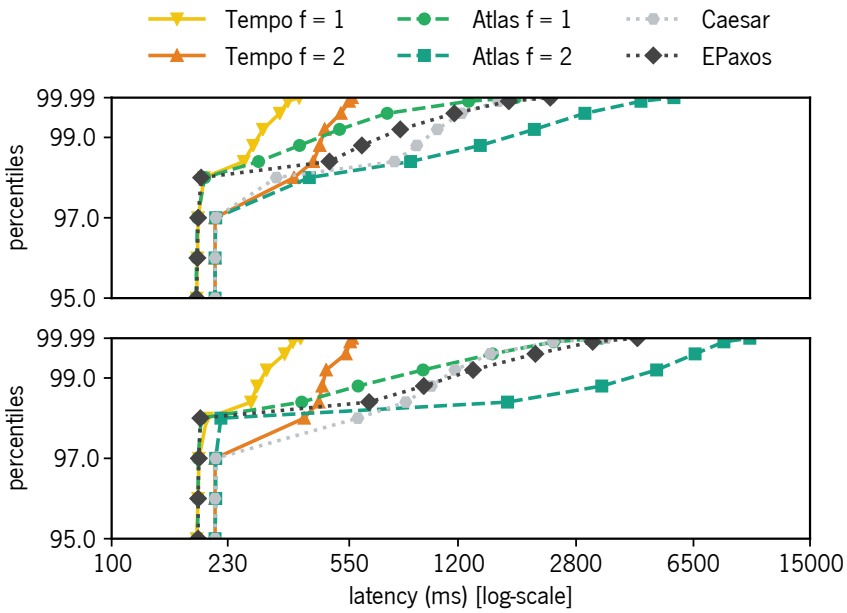


Figure 5.6: Latency percentiles with 5 sites and 256 (top) and 512 clients (bottom) per site under a low conflict rate (2%).

commands using explicit dependencies, which can arbitrarily delay command execution (§4.2.3).

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 single-machine 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 5.6 accurately capture the effect of long dependency chains and are not due to a bottleneck in the execution mechanism of the protocols.

Increasing the load and contention

We now evaluate the performance of the protocols when both the client load and contention increases. This experiment, reported in Figure 5.7, 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 5.7 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 FPaxos, we only show this measure at the leader site.

As seen in Figure 5.7, the leader in FPaxos quickly becomes a bottleneck when the load increases since it has to broadcast each command to all the processes. For this reason, FPaxos 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 at the leader reaches 95% usage. The fact that the leader can be a bottleneck in leader-based protocol has been reported by several prior works [11, 62, 68, 69].

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 $f = 2$[8]. As observed in the heatmap (bottom of Figure 5.7), `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 5.7). 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 $f = 2$ (67K ops/s). As before, EPaxos performance (omitted from Figure 5.7) lies between `Atlas` $f = 1$ and $f = 2$.

As we mentioned in §4.2.3, Caesar exhibits inefficiencies even in its commit protocol. For this reason, in Figure 5.7 we study the performance of Caesar in an ideal scenario where commands are executed as soon as they are committed. We denote this version by Caesar*. Caesar's performance is capped respectively at 104K ops/s with 2% conflicts and 32K ops/s with 10% conflicts. This performance decrease is due to Caesar's blocking mechanism (§4.2.3) and is in line with the results reported in [10].

`Tempo` delivers the maximum throughput of 230K ops/s. This value is independent of the conflict rate and fault-tolerance level (i.e., $f \in \{1, 2\}$). Moreover, it is 4.3-5.1x better than FPaxos and 1.8-3.4x better than `Atlas`. Saturation occurs with 16K clients per site, when the CPU usage reaches 95%. At this point, network utilization is roughly equal to 80%. Latency in the protocol is almost unaffected until saturation.

Increasing the number of sites

We now study a scenario in which the service expands to new locations to serve new clients. The experiment in Figure 5.8 considers 3 to 11 sites, with 256 clients per site under a low conflict rate of 2%. As before, we deploy the FPaxos leader in Ireland.

[8]We claim in [24] that in some situations `Atlas` $f = 2$ is better than `Atlas` $f = 1$. This was due to an optimization called *Reducing dependencies in the slow path* [24] which was incorrectly implemented when the set of dependencies was compressed using a vector clock (§5.2). This thesis simply disregards this optimization.

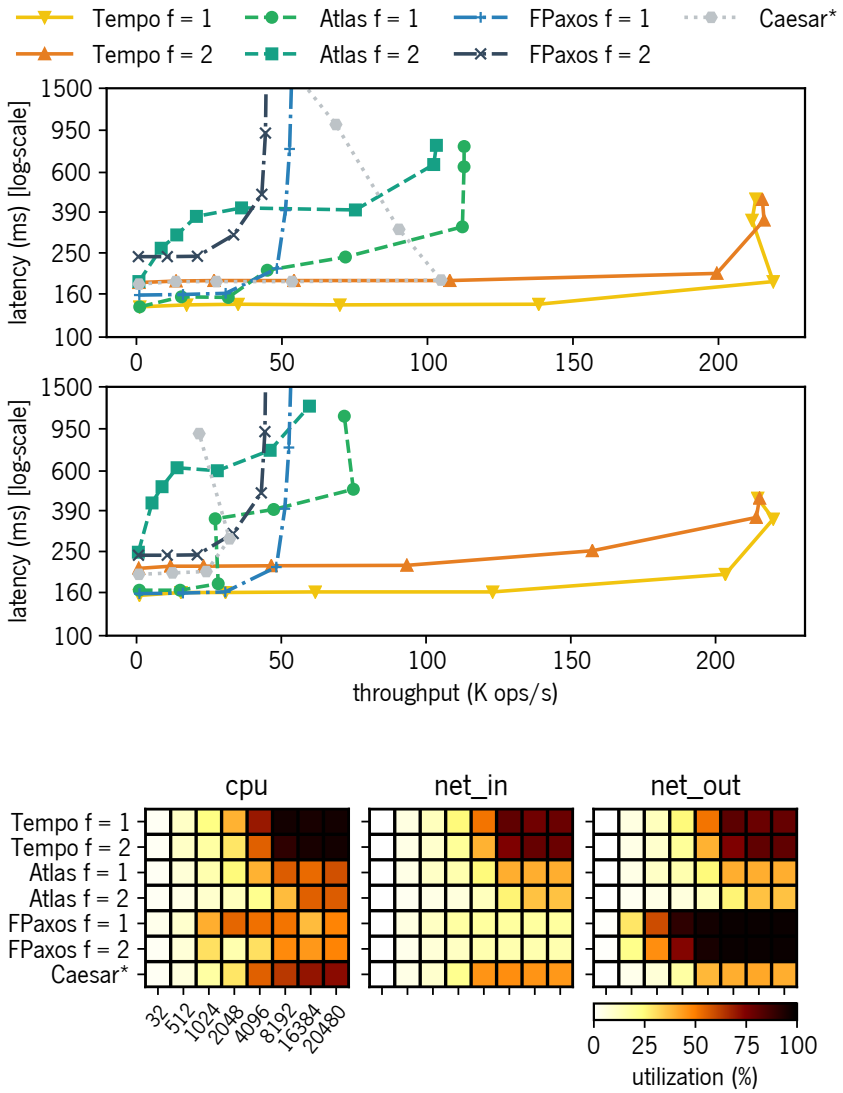


Figure 5.7: Throughput and latency with 5 sites as the load increases from 32 to 20480 clients per site under a low (2% – top) and moderate (10% – bottom) conflict rate. The heatmap shows the hardware utilization when the conflict rate is 2%.

First, note that $r = 5$ is roughly equivalent to the average bars in Figure 5.5. The only noticeable difference between the two is $\texttt{Atlas}\ f = 2$: because in Figure 5.5 we deploy two times more clients (i.e. 512 per site instead of 256), the size of the strongly connected components (§4.2.3) is higher in that experiment, which results in higher average latencies. In Figure 5.8, with $r = 5$, EPaxos performance is similar to that of $\texttt{Atlas}\ f = 1$ and $\texttt{Tempo}\ f = 1$ as the protocols have the same fast quorum size. However, due to its large fast quorums of size $\lfloor \frac{3r}{4} \rfloor$, as r increases, we observe that EPaxos becomes the protocol offering the worst performance. For example, with $r = 11$, EPaxos offers a latency 1.4x higher than $\texttt{Atlas}\ f = 1$, 1.2x higher than $\texttt{Atlas}\ f = 2$, 2x higher than $\texttt{Tempo}\ f = 1$ and 1.8x higher than $\texttt{Tempo}\ f = 2$.

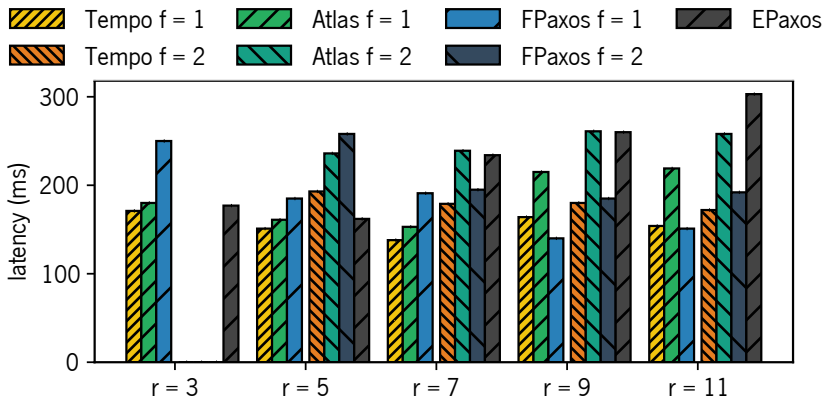


Figure 5.8: Average latency with 256 clients per site under a low conflict rate (2%) while increasing the number of sites from 3 to 11.

FPaxos performance is closely tied to which sites are considered and which of these sites is chosen as the leader. To better understand this, consider what happens when we move from $r = 7$ to $r = 9$ by focusing on Table 5.1. With $r = 7$, the closest sites to Ireland, the site where the FPaxos leader is located, are Canada and Northern Virginia (both roughly 70ms away). With $r = 9$, two new sites are added, one of them being Stockholm. Note, however, that this site is only 35ms away from Ireland. For this reason, FPaxos $f = 1$ phase 2, which requires contacting only a single site, is two times faster with $r = 9$ than with $r = 7$. As a result, the protocol can offer a average latency lower than the other protocols in this setting. With $f = 2$, the FPaxos leader has to contact, not one, but two additional processes in FPaxos $f = 2$ phase 2. Although one of these processes is nearby (Stockholm), The distance to the second closest site to Ireland did not change when we moved from $r = 7$ to $r = 9$, which is explains why FPaxos $f = 2$ offers a similar latency in both settings. This was not the case when we moved from $r = 5$ to $r = 7$, where the improvement in the performance of FPaxos $f = 2$ is quite clear. In this case, the added site responsible for the latency reduction in FPaxos $f = 2$ is Northern Virginia.

As reported before in Figure 5.5, FPaxos remains significantly more unfair than the remaining protocols. For example, with $r = 11$, FPaxos $f = 1$ provides 75ms at the leader site, but this latency more than triples at Singapore (241ms), São Paulo (235ms), Hong Kong (275ms) and Tokyo (260ms). On the other hand, leaderless protocols satisfy clients much more uniformly. For example, for the above sites, the latencies offered by `Tempo` $f = 1$ are respectively 169ms, 193ms, 203ms, 166ms and 149ms.

Availability under failures

Figure 5.9 depicts an experiment demonstrating that leaderless protocols are inherently more available than a leader-driven protocol. The experiment compares `Atlas` and FPaxos across 3 Google Cloud Platform (GCP) sites: Taiwan, Finland and South Carolina[9]. Such configuration tolerates a single site

[9]Our implementation of the protocols in **github.com/vitorenesduarte/fantoch** does not include the recovery mechanism of each protocol. For this reason, we report the experiment that was done in the `Atlas` publication [24] using our Erlang implementation of `Atlas`.

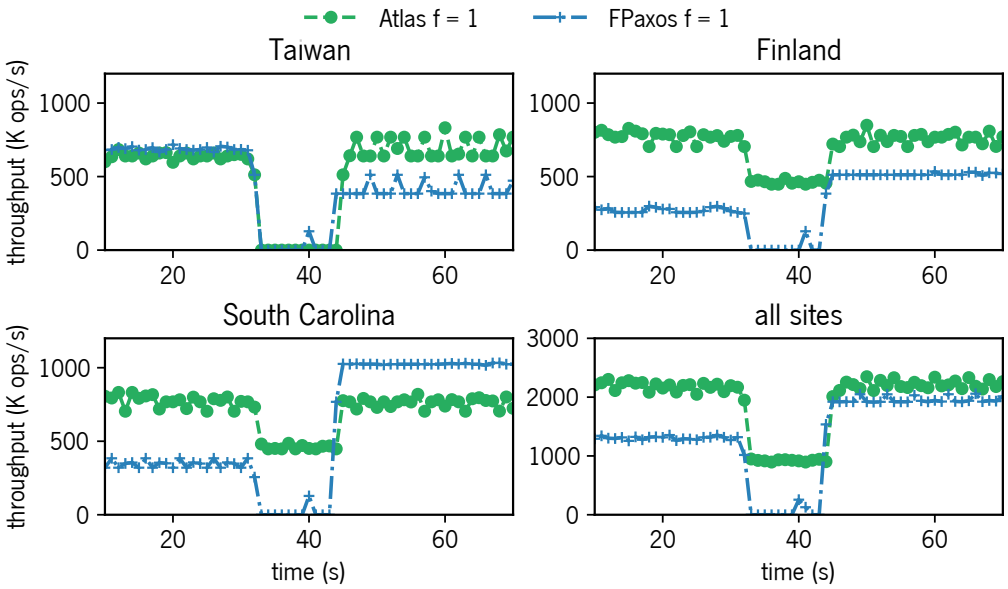


Figure 5.9: The impact of a failure on the throughput of Atlas and FPaxos (3 sites, $f = 1$).

failure, so FPaxos is the same as Paxos. We do not evaluate the remaining protocols (Tempo, EPaxos and Caesar) as their availability guarantees are similar to those of Atlas in this configuration. Each site hosts 128 clients. Half of the clients issue commands targeting key 0 and the other half issue commands targeting a unique key per client. Hence, commands by clients in the first half conflict with each other, while commands by clients in the second half commute with all commands by a different client.

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

As shown in Figure 5.9, Paxos blocks during the recovery time. In contrast, Atlas keeps executing commands, albeit at a reduced throughput. The drop in throughput happens largely because the clients issuing commands on key 0 (50% of all clients) collect as dependencies some of the commands being recovered (those that also access key 0). The execution of the former commands then blocks until the latter are recovered. In contrast, the clients at non-failed sites issuing commands with per-client keys continue to operate as normal. Since commands by these clients commute with those by other clients, their execution never blocks on the commands being recovered. This means that these clients operate without disruption during the whole experiment.

The bottom right plot contains the aggregate throughput of the system. Before failure, Atlas is almost two times faster than Paxos, and operates consistently better during the whole experiment. Note,

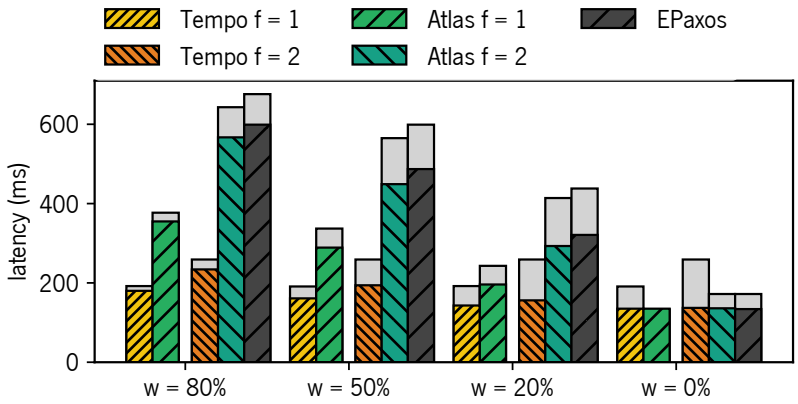


Figure 5.10: YCSB performance for update-heavy (80% writes), balanced (50% writes), read-heavy (20% writes) and read-only (0% writes) workloads, with $r = 7$ sites with NFR optimization is enabled. The latency provided by protocols when the NFR optimization is not enabled is obtained by also considering the light gray bar on top of each colored bar.

however, that Paxos has a slightly higher throughput at the leader (Taiwan) before the crash, and at the new leader (South Carolina) after recovery. This is due to the delay between committing and executing commands in Atlas.

Key-value store service

We now compare Tempo, Atlas and EPaxos when the protocols are applied to a replicated key-value store (KVS) service. When accessing a KVS record stored under key k, a client executes either command $read(k)$ to fetch its content, or $write(k, v)$ to update it to value v. To benchmark the performance of the replicated KVS we use the Yahoo! Cloud Serving Benchmark (YCSB) [74]. We apply four types of workloads, each with a different mix of $read/write$ operations: update-heavy ($\mathtt{w} = 80\%$), balanced ($\mathtt{w} = 50\%$), read-heavy ($\mathtt{w} = 20\%$), and read-only ($\mathtt{w} = 0\%$), where w represents the percentage of write commands. The KVS contains 10^6 records and all workloads select records following a Zipfian distribution with the default YCSB skew ($\mathtt{zipf} = 0.99$).

In this experiment, Tempo with $f \in \{1, 2\}$, Atlas with $f \in \{1, 2\}$ and EPaxos are deployed over 7 sites. At each site running the benchmark we execute 256 YCSB client threads. Colored bars represent the latency provided protocols when the NFR optimization (§3.2.5 and §4.2.9) is enabled. As pointed out in §3.2.5 and §4.2.9, this optimization accelerates the execution of $read$ commands and reduces their impact in the protocol stack. With the light gray bar on top of each colored bar we obtain the latency provided by protocols when the NFR optimization is not enabled.

In the update-heavy workload without NFR, EPaxos offers 676ms whereas Atlas offers 377ms when $f = 1$ and 643ms when $f = 2$. Although EPaxos and Atlas $f = 2$ have the same fast quorum size with $r = 7$, due to its more flexible fast-path condition, Atlas is able to provide a slightly better performance than EPaxos: Atlas $f = 2$ takes the fast path for 58% of commands, while EPaxos does so in 43% of cases. Tempo $f = 2$ improves on this and is able to take the fast path for 77% of commands, offering

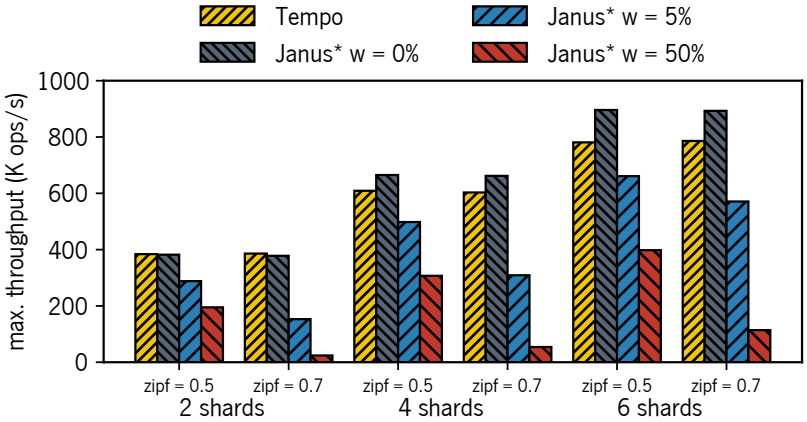


Figure 5.11: Maximum throughput with 3 sites per shard under low ($\texttt{zipf} = 0.5$) and moderate contention ($\texttt{zipf} = 0.7$). Three workloads are considered for Janus*: 0% writes (its best-case scenario), 5% writes and 50% writes.

259ms. This is 2.5x faster than Atlas $f = 2$ and 2.6x faster than EPaxos. Tempo $f = 1$ offers 192ms, which is 2x faster than Atlas $f = 1$ and 3.5x faster than EPaxos.

Increasing the percentage of read operations improves the performance of Atlas and EPaxos the protocols because reads do not conflict with other reads. In the read-only workload the performance is simply determined by the quorum size, since the protocols always take the fast path. In this case, both EPaxos and Atlas $f = 2$ offer 172ms, while Atlas $f = 1$, which has a smaller fast quorum, offers 133ms. As mentioned in §4.2.3, Tempo does not distinguish between reads and writes. For this reason, the performance of the protocol is the same in all four workloads.

With the NFR optimization, Tempo $f = 1$, Tempo $f = 2$, Atlas $f = 1$, Atlas $f = 2$ and EPaxos reduce their latency by up to 30%, 48%, 20%, 30% and 27%, respectively. The highest speedup occurs for Tempo as, without NFR, the protocol does not optimize the execution of reads (§4.2.3). In the read-only workload, all the protocols execute commands after a single round trip to the closest majority. In this case, NFR allows Tempo $f = 1$, Tempo $f = 2$, Atlas $f = 2$ and EPaxos to match the performance of vanilla Atlas $f = 1$, while maintaining their higher fault-tolerance level. Overall, Tempo $f = 1$, Tempo $f = 2$, Atlas $f = 1$ and Atlas $f = 2$ with NFR are up 3.5x, 2.9x, 2.2x and 1.5x faster than vanilla EPaxos, respectively.

5.5 Partial Replication Deployment

We now consider a partial replication setting and compare Tempo with Janus* using the YCSB+T benchmark [75]. 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 10^6 keys and is replicated at 3 sites (Ireland, N. California and Singapore) emulated in our cluster. Clients submit commands that access two keys picked at random following the YCSB access pattern (a zipfian distribution). In Figure 5.11 we show the maximum throughput for both Tempo and Janus* under low ($\texttt{zipf} = 0.5$) and moderate contention ($\texttt{zipf} = 0.7$).

For Janus*, we consider 3 YCSB workloads that vary the percentage of *write* commands (denoted by w): read-only (w = 0%, YCSB workload C), read-heavy (w = 5%, YCSB workload B), and update-heavy (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 (§4.2.3), 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 (§4.2.5). 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 5.6 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.

6

Conclusions and Future Work

In this thesis we have presented Atlas and Tempo, two SMR protocols tailored to planet-scale systems. The two protocols follow a leaderless approach, ordering commands in a fully decentralized manner and thus offering similar quality of service to all clients. These are the first leaderless SMR protocols parameterized with the number of allowed failures f. Both protocols leverage the fact that concurrent site failures are rare in planet-scale systems, and are thus optimized for small values of f. Atlas and Tempo employ small fast quorums of size $\lfloor \frac{r}{2} \rfloor + f$ and offer a flexible fast-path condition that allows a high percentage of operations to be processed within a single round trip. When $f = 1$, the protocols always take the fast path and the fast quorum is a plain majority. Moreover, the protocols employ an optimization called non-fault-tolerant reads that allows for low-latency linearizable reads with a low impact in the protocol stack.

Following EPaxos, Atlas orders commands based on explicit dependencies. In theory, EPaxos-like protocols do not ensure liveness, and in practice they offer high tail latencies. 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.

In this thesis we focused on simplifying and improving the performance of leaderless protocols. Below we list some limitations of our work and possible research avenues that may allow leaderless protocols to fulfil their potential and finally be adopted by industry practitioners [76].

Linearizable multi-key reads As mentioned in §4.2.3, Tempo does not currently distinguish between reads and writes (with the exception of the NFR optimization §4.2.9, which only applies to single-key reads). Multi-key reads can be natively supported in Tempo if, instead of maintaining a single clock per key/partition, we maintain two: one for reads, and another for writes [77]. Such strategy would be similar to how we have implemented the dependency tracking mechanism of Atlas that also distinguishes between reads and writes (§5.2.1).

Local sequentially consistent reads (LSCR) Many real-world workloads are read-dominated. Common SMR protocols that build a total order (like Paxos [7], Raft [8] and Zab [47]) can be adjusted to allow serving reads locally, without incurring a wide-area round trip. The resulting reads are not linearizable, but *sequentially consistent* [78, 79], i.e., they may read stale data. This is acceptable for many applications [80], and thus, local reads are crucial to the scalability of practical systems such as ZooKeeper [2] and etcd [81].

Allowing LSCR in leaderless protocols like EPaxos [11], `Atlas` or `Tempo` is challenging because these protocols may deliver commuting write commands in different orders at different replicas. As a consequence, reading locally from a replica may violate sequential consistency.

We believe that dependency-based protocols like EPaxos and `Atlas` cannot be easily modified to support LSCR without great performance costs. These protocols would have to build total order, which would essentially require setting the conflict rate to 100%. This would result in no fast paths and high tail latencies due to the issues in their execution mechanism (§4.2.3).

On the other hand, timestamp-based protocols like `Tempo` can be more easily modified to support LSCR: if a replica wants to serve a read at timestamp t, it is enough to delay the execution until t becomes stable at all keys/partitions accessed by the read. For keys/partitions to become stable in a timely fashion in `Tempo`, replicas should periodically issue detached promises. However, this may result in a fast-path ratio decrease.

Tiny fast quorums `Tempo` can be modified in order to use fast-quorums of size $2f$. In this case, Property 4.1 and Theorem 4.1 would have to be modified to require $f+1$ and $r-f$ processes, respectively: this would still ensure that the sets of processes used to compute a timestamp and to decide when a timestamp is stable always intersect. Further investigations are required to understand if smaller quorums required for committing a command can compensate for larger quorums required for executing them.

Bibliography

[1] João M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. url: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[2] Patrick Hunt et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems". In: *USENIX Annual Technical Conference (USENIX ATC)*. 2010 (cit. on pp. 1, 87).

[3] Michael Burrows. "The Chubby Lock Service for Loosely-Coupled Distributed Systems". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2006 (cit. on pp. 1, 3, 11, 15, 16).

[4] James C. Corbett et al. "Spanner: Google's Globally-Distributed Database". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2012 (cit. on pp. 1–3, 6, 11, 15, 16, 28, 68–71).

[5] Fred B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial". In: *ACM Comput. Surv.* (1990) (cit. on pp. 1, 6).

[6] Maurice Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* (1990) (cit. on pp. 1, 6, 38).

[7] Leslie Lamport. "The Part-Time Parliament". In: *ACM Trans. Comput. Syst.* (1998) (cit. on pp. 1, 2, 6, 9, 16, 25, 35, 54, 68, 87).

[8] Diego Ongaro and John K. Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *USENIX Annual Technical Conference (USENIX ATC)*. 2014 (cit. on pp. 1, 6, 16, 35, 68, 87).

[9] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. "Gryff: Unifying Consensus and Shared Registers". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2020 (cit. on pp. 1, 3, 37, 48, 49, 68).

[10] Balaji Arun et al. "Speeding up Consensus by Chasing Fast Decisions". In: *International Conference on Dependable Systems and Networks (DSN)*. 2017 (cit. on pp. 1, 3, 11, 12, 16, 35, 37, 48, 49, 51, 68, 71, 79).

[11] Iulian Moraru, David G. Andersen, and Michael Kaminsky. “There Is More Consensus in Egalitarian Parliaments”. In: *Symposium on Operating Systems Principles (SOSP)*. 2013 (cit. on pp. 1–3, 11, 16, 17, 22, 23, 28, 35, 36, 48, 49, 68, 71, 72, 79, 87).

[12] Alexandru Turcu et al. “Be General and Don’t Give Up Consistency in Geo-Replicated Transactional Systems”. In: *International Conference on Principles of Distributed Systems (OPODIS)*. 2014 (cit. on pp. 1, 35).

[13] Fernando Pedone and André Schiper. “Generic Broadcast”. In: *International Symposium on Distributed Computing (DISC)*. 1999 (cit. on pp. 1, 11, 17, 35).

[14] Leslie Lamport. *Generalized Consensus and Paxos*. Tech. rep. MSR-TR-2005-33. Microsoft Research, 2005 (cit. on pp. 1–3, 11, 15, 17, 22, 35).

[15] Shuai Mu et al. “Consolidating Concurrency Control and Consensus for Commits under Conflicts”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2016 (cit. on pp. 2, 28, 38, 39, 59, 69, 71, 72).

[16] Alexander Shraer et al. “Dynamic Reconfiguration of Primary/Backup Clusters”. In: *USENIX Annual Technical Conference (USENIX ATC)*. 2012 (cit. on p. 2).

[17] Shengyun Liu et al. “XFT: Practical Fault Tolerance beyond Crashes”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2016 (cit. on pp. 2, 16, 28, 70).

[18] Seth Gilbert and Nancy A. Lynch. “Brewer’s Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services”. In: *SIGACT News* (2002) (cit. on pp. 3, 6, 28).

[19] Tuanir França Rezende and Pierre Sutra. “Leaderless State-Machine Replication: Specification, Properties, Limits”. In: *International Symposium on Distributed Computing (DISC)*. 2020 (cit. on pp. 3, 11, 36, 37, 49).

[20] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. “Mencius: Building Efficient Replicated State Machine for WANs”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2008 (cit. on pp. 3, 35, 68).

[21] Jiaqing Du et al. “Clock-RSM: Low-Latency Inter-datacenter State Machine Replication Using Loosely Synchronized Physical Clocks”. In: *International Conference on Dependable Systems and Networks (DSN)*. 2014 (cit. on pp. 3, 37, 68).

[22] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* (1978) (cit. on pp. 3, 6, 37).

[23] James A. Cowling and Barbara Liskov. “Granola: Low-Overhead Distributed Transaction Coordination”. In: *USENIX Annual Technical Conference (USENIX ATC)*. 2012 (cit. on pp. 3, 37, 69).

[24] Vitor Enes et al. “State-Machine Replication for Planet-Scale Systems”. In: *European Conference on Computer Systems (EuroSys)*. 2020 (cit. on pp. 4, 73, 75, 79, 81).

[25] Vitor Enes et al. “Efficient Replication via Timestamp Stability”. In: *European Conference on Computer Systems (EuroSys)*. 2021 (cit. on p. 4).

[26] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. “Impossibility of Distributed Consensus with One Faulty Process”. In: *J. ACM* (1985) (cit. on p. 5).

[27] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. “Consensus in the Presence of Partial Synchrony”. In: *J. ACM* (1988) (cit. on p. 6).

[28] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. “The Weakest Failure Detector for Solving Consensus”. In: *J. ACM* (1996) (cit. on pp. 6, 10, 56).

[29] Daniel Abadi. “Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story”. In: *Computer* (2012) (cit. on p. 6).

[30] Martin Kleppmann. “A Critique of the CAP Theorem”. In: *arXiv CoRR* abs/1509.05393 (2015). url: http://arxiv.org/abs/1509.05393 (cit. on p. 6).

[31] Haonan Lu et al. “Existential consistency: measuring and understanding consistency at Facebook”. In: *Symposium on Operating Systems Principles (SOSP)*. 2015 (cit. on p. 6).

[32] Rebecca Taft et al. “CockroachDB: The Resilient Geo-Distributed SQL Database”. In: *International Conference on Management of Data (SIGMOD)*. 2020 (cit. on p. 6).

[33] Leslie Lamport. “Paxos Made Simple”. In: *SIGACT News* (2001) (cit. on pp. 6, 9).

[34] Robbert van Renesse and Deniz Altinbuken. “Paxos Made Moderately Complex”. In: *ACM Comput. Surv.* (2015) (cit. on p. 6).

[35] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. “Flexible Paxos: Quorum Intersection Revisited”. In: *International Conference on Principles of Distributed Systems (OPODIS)*. 2016 (cit. on pp. 6, 9, 16, 22, 25, 28, 35, 40, 45, 54, 71).

[36] Heidi Howard. *Distributed consensus revised*. Tech. rep. UCAM-CL-TR-935. PhD Thesis. 2019 (cit. on p. 6).

[37] Heidi Howard and Richard Mortier. “Paxos vs Raft: have we reached consensus on distributed consensus?” In: *Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC@EuroSys)*. 2020 (cit. on p. 6).

[38] Michael Whittaker et al. “SoK: A Generalized Multi-Leader State Machine Replication Tutorial”. In: *JSys* (2021) (cit. on p. 6).

[39] Rachid Guerraoui and André Schiper. “Genuine Atomic Multicast in Asynchronous Distributed Systems”. In: *Theor. Comput. Sci.* (2001) (cit. on pp. 12, 59, 69).

[40] Leslie Lamport. “Fast Paxos”. In: *Distributed Computing* (2006) (cit. on p. 22).

[41] Leslie Lamport. “Lower Bounds for Asynchronous Consensus”. In: *Distributed Computing* (2006) (cit. on p. 28).

[42] Henrique Moniz et al. “Blotter: Low Latency Transactions for Geo-Replicated Storage”. In: *International Conference on World Wide Web (WWW)*. 2017 (cit. on p. 28).

[43] Iulian Moraru. *Egalitarian Distributed Consensus*. Tech. rep. CMU-CS-14-133. PhD Thesis. Carnegie Mellon University, 2014 (cit. on p. 28).

[44] Muhammed Uluyol et al. “Near-Optimal Latency Versus Cost Tradeoffs in Geo-Distributed Storage”. In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2020 (cit. on p. 28).

[45] Pierre Sutra. “On the correctness of Egalitarian Paxos”. In: *Inf. Process. Lett.* (2020) (cit. on p. 28).

[46] Heidi Howard, Aleksey Charapko, and Richard Mortier. “Fast Flexible Paxos: Relaxing Quorum Intersection for Fast Paxos”. In: *International Conference on Distributed Computing and Networking (ICDCN)*. 2021 (cit. on p. 28).

[47] Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. “Zab: High-performance broadcast for primary-backup systems”. In: *International Conference on Dependable Systems and Networks (DSN)*. 2011 (cit. on pp. 35, 87).

[48] Brian M. Oki and Barbara Liskov. “Viewstamped Replication: A General Primary Copy”. In: *Symposium on Principles of Distributed Computing (PODC)*. 1988 (cit. on p. 35).

[49] Carlos Eduardo Benevides Bezerra, Fernando Pedone, and Robbert van Renesse. “Scalable State-Machine Replication”. In: *International Conference on Dependable Systems and Networks (DSN)*. 2014 (cit. on pp. 35, 52).

[50] Sebastiano Peluso et al. “Making Fast Consensus Generally Faster”. In: *International Conference on Dependable Systems and Networks (DSN)*. 2016 (cit. on p. 35).

[51] Ailidani Ailijiang et al. “Multileader WAN Paxos: Ruling the Archipelago with Fast Consensus”. In: *arXiv CoRR* abs/1703.08905 (2017). url: http://arxiv.org/abs/1703.08905 (cit. on p. 35).

[52] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. “DPaxos: Managing Data Closer to Users for Low-Latency and Mobile Applications”. In: *International Conference on Management of Data (SIGMOD)*. 2018 (cit. on p. 35).

[53] Jialin Li et al. “Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2016 (cit. on p. 36).

[54] Huynh Tu Dang et al. “NetPaxos: Consensus at Network Speed”. In: *Symposium on Software Defined Networking Research (SOSR)*. 2015 (cit. on p. 36).

[55] Cheng Wang et al. “APUS: Fast and Scalable Paxos on RDMA”. In: *Symposium on Cloud Computing (SoCC)*. 2017 (cit. on p. 36).

[56] JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi. “Partial Database Replication using Epidemic Communication”. In: *International Conference on Distributed Computing Systems (ICDCS)*. 2002 (cit. on p. 38).

[57] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. “P-Store: Genuine Partial Replication in Wide Area Networks”. In: *Symposium on Reliable Distributed Systems (SRDS)*. 2010 (cit. on pp. 38, 69).

[58] Raluca Halalai et al. “ZooFence: Principled Service Partitioning and Application to the ZooKeeper Coordination Service”. In: *Symposium on Reliable Distributed Systems (SRDS)*. 2014 (cit. on p. 39).

[59] Robert Kallman et al. “H-Store: A High-Performance, Distributed Main Memory Transaction Processing System”. In: *Proc. VLDB Endow.* (2008) (cit. on p. 39).

[60] Haonan Lu et al. “The SNOW Theorem and Latency-Optimal Read-Only Transactions”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2016 (cit. on p. 39).

[61] Alexander Thomson and Daniel J. Abadi. “The Case for Determinism in Database Systems”. In: *Proc. VLDB Endow.* (2010) (cit. on p. 39).

[62] Michael Whittaker et al. “Bipartisan Paxos: A Modular State Machine Replication Protocol”. In: *arXiv CoRR* abs/2003.00331 (2020). url: https://arxiv.org/abs/2003.00331 (cit. on pp. 48, 49, 68, 79).

[63] Mahesh Balakrishnan et al. “CORFU: A Shared Log Design for Flash Clusters”. In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2012 (cit. on p. 68).

[64] Jianjun Zheng et al. “PaxosStore: High-availability Storage Made Practical in WeChat”. In: *Proc. VLDB Endow.* (2017) (cit. on p. 68).

[65] Mahesh Balakrishnan et al. “Tango: Distributed Data Structures over a Shared Log”. In: *Symposium on Operating Systems Principles (SOSP)*. 2013 (cit. on p. 68).

[66] Daniel Peng and Frank Dabek. “Large-scale Incremental Processing Using Distributed Transactions and Notifications”. In: *Symposium on Operating Systems Design and Implementation (OSDI)*. 2010 (cit. on p. 68).

[67] Alexander Thomson et al. “Calvin: Fast Distributed Transactions for Partitioned Database Systems”. In: *International Conference on Management of Data (SIGMOD)*. 2012 (cit. on p. 68).

[68] Antonios Katsarakis et al. “Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol”. In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2020 (cit. on pp. 68, 79).

[69] Marios Kogias and Edouard Bugnion. “HovercRaft: Achieving Scalability and Fault-tolerance for microsecond-scale Datacenter Services ”. In: *European Conference on Computer Systems (EuroSys)*. 2020 (cit. on pp. 68, 79).

[70] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. “The Google File System”. In: *Symposium on Operating Systems Principles (SOSP)*. 2003 (cit. on p. 69).

[71] Brian M. Oki and Barbara Liskov. “Viewstamped Replication: A General Primary Copy”. In: *Symposium on Principles of Distributed Computing (PODC)*. 1988 (cit. on p. 69).

[72] Tim Kraska et al. “MDCC: Multi-Data Center Consistency”. In: *European Conference on Computer Systems (EuroSys)*. 2013 (cit. on pp. 69, 71).

[73] Irene Zhang et al. “Building Consistent Transactions with Inconsistent Replication”. In: *Symposium on Operating Systems Principles (SOSP)*. 2015 (cit. on pp. 69, 71).

[74] Brian F. Cooper et al. “Benchmarking Cloud Serving Systems with YCSB”. In: *Symposium on Cloud Computing (SoCC)*. 2010 (cit. on pp. 72, 74, 83).

[75] Akon Dey et al. “YCSB+T: Benchmarking Web-scale Transactional Databases”. In: *International Conference on Data Engineering Workshops (ICDEW)*. 2014 (cit. on pp. 74, 84).

[76] Cassandra. *Cassandra Enhancement Proposal (CEP-15): General Purpose Transactions*. url: https://cwiki.apache.org/confluence/display/CASSANDRA/CEP-15%3A+General+Purpose+Transactions (cit. on p. 86).

[77] Fedor Ryabinin. *Personal Communication* (cit. on p. 86).

[78] Leslie Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *IEEE Trans. Computers* (1979) (cit. on p. 87).

[79] Kfir Lev-Ari et al. “Composing ordered sequential consistency”. In: *Inf. Process. Lett.* (2017) (cit. on p. 87).

[80] Ruoming Pang et al. “Zanzibar: Google’s Consistent, Global Authorization System”. In: *USENIX Annual Technical Conference (USENIX ATC)*. 2019 (cit. on p. 87).

[81] etcd. *etcd GitHub repository*. url: https://github.com/coreos/etcd (cit. on p. 87).