Leaderless state-machine replication : from fail-stop to Byzantine failures

Tuanir Franca Rezende

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2021IPPAS016

Thèse de doctorat

Leaderless State-Machine Replication: From Fail-stop to Byzantine Failures

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom SudParis

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 17 Decembre 2021, par

TUANIR FRANÇA REZENDE

Composition du Jury :

Pierre SENS Professeur, Sorbonne Université	Président
Étienne RIVIÈRE Professeur, Université catholique de Louvain	Rapporteur
Janna Burman Maître de Conférence HDR, LRI	Rapporteuse
Maryline LAURENT, Télécom SudParis (SAMOVAR) Professeure	Examinateur
Denis CONAN Maître de Conférence, HDR, Télécom SudParis (SAMOVAR)	Directeur de thèse
Pierre SUTRA Maître de Conférence, Télécom SudParis (SAMOVAR)	Co-directeur de thèse

To my parents Nilcea and Tuanir

Abstract

Modern distributed services are expected to be highly available, as our societies have been growing increasingly dependent on them. The common way to achieve high availability is through the replication of data in multiple service replicas. In this way, the service remains operational in case of failures as clients can be relayed to other working replicas. In distributed systems, the classic technique to implement such fault-tolerant services is called State-Machine Replication (SMR), where a service is defined as a deterministic state-machine and each replica keeps a local copy of the machine. To guarantee that the service remains consistent, replicas coordinate with each other and agree on the order of transitions to be applied to their copies of the state-machine.

The replication performed by modern Internet services spans across several geographical locations (geo-replication). This allows for increased availability and low latency, since clients can communicate with the closest geo-graphical replica. Due to their reliance on a leader replica, classical SMR protocols offer limited scalability and availability under this setting. To solve this problem, recent protocols follow instead a leaderless approach, in which each replica is able to make progress using a quorum of its peers. These new leaderless protocols are complex and each one presents an ad-hoc approach to leaderlessness.

The first contribution of this thesis is a framework that captures the essence of Leaderless State-Machine Replication (Leaderless SMR) and the formalization of some of its limits. Due to the increasingly sensitive nature of replicated services, leveraging simple benign failures is no longer enough. Recent research is headed towards developing protocols that support arbitrary behavior of some replicas (Byzantine failures) and that also thrive in a geo-replicated environment. An example of this new type of sensitive replicated services that has been the focus of a lot of research are blockchains. Blockchains are powered by Byzantine replication protocols adapted to work over hundreds or even thousands of replicas. When the membership control over such replicas is open, that is, anyone can run a replica, we say the blockchain is permissionless. In the converse case, when the membership is controlled by a set of known entities like companies, we

say the blockchain is permissioned. When such Byzantine protocols follow the classic leader-driven approach they suffer from scalability and availability issues, similarly to their non-byzantine counterparts. In the second part of this thesis, we adapt our framework to support Byzantine failures and present the first framework for Byzantine Leaderless SMR. Furthermore, we show that when properly instantiated it allows to sidestep the scalability problems in leader-driven Byzantine SMR protocols for permissioned blockchains.

Acknowledgements

I would like to thank my advisors Pierre Sutra and Denis Conan for their patience and fruitful discussions. I would like to thank my parents Tuanir and Nilcea for their inexorable love and for giving me all the fundamentals necessary to become who I am today. I would like to express my gratitude to all my Brazilian friends, that even from afar kept pouring support: Alisson, Camila, Alvaro, Augusto, Rodrigo, Ludmila, Barbara. I'd like to thank all of those that helped a lot at the beginning of this quest: Nabila, Monika, Patryk, Imane, Erika. Special thanks to fellow PhD students Alexis, Boubacar and Pedro. I'd like to thank all in the Fondation Suisse with special thanks to those that were there in the bitter end: Rafael, Joanna, Daniel, Jonas, Victor, Sophie, Blanka, Steph, Mdm Corrado, Miriam, Naomi (for boosting my confidence), Timofei.

Preface

This thesis is the result of my PhD research [1] under the supervision of Prof. Pierre Sutra and Prof. Denis Conan. A part of it was published in international conferences, and the last part is under submission.

- *Fast State-Machine Replication via Monotonic Generic Broadcast* (technical report [132])

  Authors: Vitor Enes, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, Pierre Sutra.

- *Leaderless State-Machine Replication: Specification, Properties, Limits* [116] [2]

  34th International Symposium on Distributed Computing (DISC 2020).

  Authors: Tuanir França Rezende, Pierre Sutra.

- *State-machine replication for planet-scale systems* [62].

  EuroSys 2020.

  Authors: Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, Pierre Sutra.

- *Scalable Byzantine Fault-Tolerance through Leaderless State-Machine Replication* [115] [3]

  FOCODILE 2021 - 2nd International Workshop on Foundations of Consensus and Distributed Ledgers.

  Author: Tuanir França Rezende.

[1] This research is partly funded by the ANR RainbowFS project and the H2020 CloudButton project.

[2] Video presentation available at: https://www.youtube.com/watch?v=aA751sh7Sf4

[3] Video presentation available at: https://www.youtube.com/watch?v=oTxnqI5Qwc4

- *Scaling Permissioned Blockchains via Byzantine Leaderless SMR* (under submission)

  Authors: Tuanir França Rezende, Pierre Sutra.

Contents

Contents **ix**

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
1.1 Research Context 1
1.2 Motivation and research problems 2
1.3 Thesis Contributions 3
1.4 Thesis Overview 3

2 Background and Motivation **5**
2.1 Fail-stop 6
2.1.1 System model 6
2.1.2 Classic SMR 6
2.1.3 Generic SMR 10
2.1.4 Leaderless SMR 11
2.2 Byzantine Fault Tolerance 12
2.2.1 System Model 13
2.2.2 Byzantine Fault-Tolerant State-Machine Replication (BFT SMR) 14
2.2.3 Key Notions 14
2.2.4 Historical Solutions 17
2.2.5 Practical Byzantine Fault-Tolerance (PBFT) 18
2.2.6 700BFT 20
2.2.7 Query/Update (Q/U) and Hybrid Quorum (HQ) 20
2.3 Blockchain 21
2.3.1 Bitcoin 23
2.3.2 Algorand 25

2.3.3 Stellar . . . 26
2.3.4 HotStuff . . . 27
2.3.5 SBFT . . . 27
2.3.6 Pompe . . . 28
2.4 Motivation . . . 29

3 Leaderless SMR **33**
3.1 Introduction . . . 33
3.2 Leaderless SMR . . . 34
3.2.1 Definition . . . 34
3.2.2 Deciding commands . . . 36
3.2.3 Examples . . . 38
3.2.4 Core properties . . . 40
3.3 The ROLL theorem . . . 41
3.3.1 Optimality . . . 43
3.4 Chaining effect . . . 44
3.4.1 Notion of chain . . . 44
3.4.2 A measure of asynchrony . . . 45
3.4.3 Result statement . . . 45
3.5 Discussion . . . 47
3.6 Related work . . . 48

4 Byzantine Leaderless SMR **51**
4.1 Introduction . . . 51
4.1.1 Key Observations . . . 51
4.1.2 Primer on the results . . . 52
4.2 Byzantizing Leaderless SMR . . . 53
4.2.1 Notion of Trusted Service . . . 55
4.2.2 Algorithm . . . 57
4.3 Implementations . . . 61
4.3.1 Preliminaries . . . 61
4.3.2 À la EPaxos . . . 62
4.3.3 Wintermute . . . 66
4.3.4 Efficiently representing dependencies . . . 66
4.3.5 Protecting the dependencies representation . . . 69
4.4 Complexity Analysis . . . 74
4.4.1 Possible Extensions . . . 75
4.5 Related Work . . . 75

5 Conclusion **79**
5.1 Summary . . . 79
5.2 Future Work . . . 79

A System Model **81**
A.1 Model . . . 81
A.2 Technical Lemmas . . . 83
A.3 Proofs of Theorems 1 and 2 . . . 84

B The ROLL Theorem **89**
B.1 The Properties . . . 89
B.2 Characterizing ROLL protocols . . . 90
B.3 Proof . . . 92
B.4 Chaining effect . . . 94
B.4.1 An Inductive Reasoning . . . 94
B.4.2 Preliminaries . . . 95
B.4.3 Inductive step: $\mathfrak{P}(i) \Rightarrow \mathfrak{P}(i+1)$ . . . 96

C Linearizable objects with BLSMR **99**
C.1 Preliminaries . . . 99
C.2 Algorithm . . . 102
C.3 Correctness . . . 103

Bibliography **107**

Figures

2.1 Classic Paxos message exchange in the failure-free case. The star marks the learning of a command. 8

2.2 EPaxos normal-case. $N = 5, f = 2$. Messages and their corresponding phases are described by the following colors: **PREACCEPT**, PREACCEPTACK, ACCEPT, ACCEPTACK, *COMMIT* (dashed lines). 13

2.3 PBFT phases in the normal-case. 18

2.4 Performance comparison of EPaxos, Paxos and Mencius – *5 sites: South Carolina (SC), Finland (FI), Canada (QC), Australia (AU), Taiwan (TW, leader); 128 clients per site;* no-op *service*. 30

3.1 An example run of Leaderless SMR – *(left) processes p_1 and p_2 submit respectively the commands* $\{\mathsf{a}\}$ *and* $\{\mathsf{b},\mathsf{c},\mathsf{d}\}$*; (right) the dependencies graphs formed at the two processes.* 35

3.2 Illustration of Theorem 3 – *slow messages are omitted*. 42

3.3 Theorem 4 for $n = 5$ and $k = 7$. The chain $\mathsf{c}_7\mathsf{c}_6\mathsf{c}_5\mathsf{c}_4\mathsf{c}_3\mathsf{c}_2\mathsf{c}_1$ is formed in σ_7. Illustrating the steps S_1, M_1 and R_1 for command c_1, the prefix Γ_7 of σ_7, and the steps Γ'_7. 46

4.1 Example of quorums in the S-BQS quorum system, with $f \leq 2$, $n = 9^2$ and thus $\zeta = 2$. Each quorum (in green and purple) consists of two rows and two columns. Their intersection is shaded. 71

B.1 The symbol × represents coordinator placement 96

Tables

2.1 Performance of blockchains. *: Scalability for inter-replica latency 10ms +-1.0ms, with 0/0 payload, batch size of 400. [3]. **: Counting bitcoin full nodes [29] 31

3.1 The properties of several leaderless SMR protocols – Min stands for a minority of replicas ($\lfloor \frac{n-1}{2} \rfloor$), Maj a majority ($\lceil \frac{n+1}{2} \rceil$), and LMaj a large majority ($\lfloor \frac{3n}{4} \rfloor$). 41

4.1 Performance of a collection of BFT protocols. In the critical path, always during nice runs, from a quiescent state (i.e. best case). Δ: Latency (message delays), α: Authenticators complexity, μ: Message Complexity, l: Load. * The actual message delay for a decision will be higher depending on the byzantine consensus protocol used. 53

Chapter 1

Introduction

Modern internet services are deployed on an ever growing infrastructure comprised of several data centers each with thousands of computers, often spread all over the world. As our societies have grown increasingly dependent on such services, their unavailability [49, 113] have wide spread consequences [128]. Once limited mostly to monetary costs to their owners, service failures now affect businesses and communications world-wide. This thesis aims at advancing the state-of-the-art on the techniques used to build highly available distributed services by providing new abstractions and protocols that take into account this new planet-scale paradigm.

1.1 Research Context

To achieve high availability, distributed services often replicate their critical data in multiple replicas. In this way, if a failure occurs the service remains operational as clients can still access it through other working replicas. In distributed systems, the classic technique to implement such fault-tolerant services is called *State-Machine Replication (SMR)* [122]. It allows a set of distributed processes (replicas) to construct a linearizable [80] shared object. In SMR a service is defined as a deterministic state-machine together with a set of commands and each process maintains its own local copy of the machine. An SMR protocol coordinates the execution of commands applied to the state-machine, ensuring that the replicas stay in sync.

At the core of SMR is a consensus protocol used by the replicas to decide on a common application order of commands. In the past years, the most used consensus protocol has been Paxos [88], present in well-established systems like the lock service Chubby [35] and the distributed store Spanner [44]. Despite its

success among practitioners, Paxos has well-known limitations. Notably, its design centered around a leader replica responsible to order all clients commands. Such leader based approach is present in a wide-range of classic SMR protocols (e.g Raft [109]).

1.2 Motivation and research problems

The replication paradigm of modern distributed services is of *geo-replication*, that is, replicas are placed across several geographical locations. Geo-replication allows for increased availability and low latency, once clients can communicate with the closest geographical replica.

Due to their reliance on a leader replica, classical SMR protocols like Paxos and Raft offer limited scalability and availability under geo-replication (e.g. clients geographically far from the leader replica will suffer from high latency when interacting with the system). To address such an issue, recent protocols such as EPaxos [106] and Mencius [104] follow instead a *leaderless* approach, in which each replica is able to make progress as long as it can contact a subset of its peers. Although this new class of leaderless protocols offer a promising approach to tackle geo-replication, its wider-adoption is harmed due to their high complexity and *ad-hoc* approach to leaderlessness.

In the context of geo-replication an application that has been focus of a lot of research are blockchains. A blockchain can be seen as a distributed and replicated tamper-proof ledger of transactions. Transactions between clients are gathered in blocks, each block is cryptographically linked to a previous one, and a replication protocol resilient to arbitrary failure of replicas (*Byzantine* failures) is responsible to correctly replicate the chain of blocks among replicas. A characteristic shared by all the aforementioned SMR protocols is that they support only benign failure of replicas and cannot be used out-of-the-box in the context of blockchains.

Perhaps the most famous blockchain protocol and that sparkled the interest in the field is Bitcoin [107]. Bitcoin follows the permissionless model, where anyone is able to join the network and run a replica. Guaranteeing the consistency of the blockchain in such a model, where processes can join and leave at any point, act maliciously for their own financial gains and fake identities is notably complex and expensive (it relies on a computationally expensive byzantine replication protocol [27]). To avoid the complexities of the previous model and the costs associated with it, an alternative is to consider a different model (*permissioned*) where the membership of the blockchain is controlled by a set of

known entities like companies. In such case, the blockchains are often powered by classic Byzantine hardened state-machine replication (Byzantine SMR). When permissioned blockchain protocols follow the classic leader driven approach they suffer from scalability and availability issues, similarly to their non-byzantine counterparts.

1.3 Thesis Contributions

In this thesis we propose a framework that captures the essence of Leaderless State-Machine Replication (Leaderless SMR) and formally state some of its limits.

Further, we adapt our framework to support Byzantine failures and present the first framework for Byzantine Leaderless SMR. We show that when properly instantiated it allows one to sidestep the scalability issue of leader driven Byzantine SMR protocols, which is of interest in the context of permissioned blockchains.

1.4 Thesis Overview

Background and Motivation. In Chapter 2 we recall the principles of Classic and Generic SMR under the fail-stop model and we also present the new approach of leaderless state-machine replication. Further, we present the Byzantine fault-tolerant model and a few important key notions (e.g. Byzantine Quorum Systems). We also present a collection of Byzantine SMR protocols and introduce blockchains. Last, we present permissioned blockchains replication protocols and argue that leaderless protocols can be of interest for new protocols in both fault models. The experiments in §2.4 and the important observations taken from it are the result of our work in [132, 62].

Leaderless State-Machine Replication: Specification, Properties, Limits. In Chapter 3 we formally define Leaderless SMR and deconstruct it into basic building blocks. We present a framework that accurately captures this new class of protocols and show that different leaderless protocols can be cast to it. Lastly, we state some of their limits. This chapter is derived from our work in [116].

Byzantine Leaderless SMR. In Chapter 4 we present the first framework for Byzantine Leaderless State-Machine Replication (BLSMR) and we propose different instantiations to it. Notably, we present: *Wintermute* a protocol that has overall lower load and message complexity than common Byzantine SMR protocols. This protocol allows to sidestep scalability issues of permissioned blockchains.

The content of this chapter appeared first in [115].

Conclusion. In Chapter 5 we present our final remarks and future work.

Chapter 2

Background and Motivation

Modern Internet services commonly replicate critical data across several geographical locations using state-machine replication (SMR) [122]. Due to their reliance on a leader replica, classical SMR protocols offer limited scalability and availability in this setting. To solve this problem, recent protocols follow instead a *leaderless approach*, in which each replica is able to make progress using a subset of its peers. Further, due to the increasing sensitive nature of replicated services, leveraging simple benign failures is no longer enough. Therefore, developing and adapting such promising approach for this harsher fault model is of the essence to build new reliable protocols.

In this Chapter 2 we provide the theoretical background necessary to understand this new class of leaderless protocols by first formally defining the problem of state-machine replication under the classic scenario, that is, replicas simply present benign failure (aka *fail-stop*). We also present Generic SMR, the problem from which several leaderless protocols took inspiration from and introduce Leaderless SMR, using as example the most famous leaderless protocol: EPaxos [106].

Later, we redefine and explain SMR under a scenario where replicas can fail in a malicious manner (*Byzantine* failures). We explore this failure model more in-depth through the introduction of key notions and analysis of classic and modern protocols. We also introduce the concept of a Blockchain, a type of Byzantine resilient replicated service that embodies the issues that modern replication techniques suffer from: scalability in the context of geo-replication.

Finally, we close the chapter by introducing a discussion that encompasses the motivation of this thesis: To properly define and study the new class of Leaderless SMR protocols, such that modern large-scale geo-replication protocols can be built for the fail-stop and byzantine fault-tolerant models.

2.1 Fail-stop

2.1.1 System model

We consider the standard model of wait-free computation in a distributed message-passing system where processes [1] may fail-stop [64] (i.e. components fail by stopping or by omitting steps). In [42], the authors extend this framework to include failure detectors. Further details appear in Appendix A.

2.1.2 Classic SMR

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

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

When the properties below hold during every execution, the above construct ensures that the replicated state machine implements a linearizable [80] shared object.

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

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

Consistency: For any two processes p and q, if $log_p[i]$ and $log_q[i]$ are both not $\perp$, then they are equal.

Example (Paxos)

The baseline for SMR implementation is Paxos [88], a primary-backup style of a protocol where a primary (i.e. leader) drives the decisions. For didactic purposes

[1] Throughout this thesis we use the term *replica* and *process* interchangeably.

we follow the terminology of [89] to detail the protocol, this means we assume that processes can have different roles: proposers, acceptors and learners.

Paxos processes execute multiple ballots (i.e. rounds), which are totally ordered by a relation $<$. Each ballot has a number associated with it, typically represented as a natural number, and each ballot is coordinated by one specific proposer. Even though there is a total order among ballot numbers, the ballot execution doesn't need to follow this order, and actions in different ballots may even interleave. The common approach to implement SMR using Paxos is to consider a set of ordered Paxos instances (i.e. ordered consensus instances), where each instance is used to decide a submitted commands position in the log.

In Paxos, clients submit state-machine commands wrapped in proposal messages to *proposers*, but only a proposer that *believes itself to be the leader (or coordinator) of the consensus instance* has the rights to propose a command to the set of acceptors. Putting in context of the model presented in §2.1.1, this is achieved via the proposer querying the failure detector oracle. To propose, the coordinator must start one of its ballots, bigger than any other previously started. Its proposal will become the *decided*[2] value of the instance, iff a correct and reachable majority of acceptors for that ballot exists and no other process has the wrong idea of being the coordinator (again via querying the failure detector oracle), which could lead multiple ballots being started, preventing each other from terminating.

A ballot in Paxos is divided into two phases: the first phase, referenced in the literature as *configuration* phase or *prepare* phase, serves the purpose of identifying previously decided values and the second phase, referenced as the *accept* phase, tries to get some value decided in some ballot. The first phase is comprised of actions: *Phase1a, Phase1b* and the second one of actions: *Phase2a, Phase2b*. One must understand that actions *Phase1a, Phase2a* initiate each phase, while actions *Phase1b, Phase2b* may be seen as responses to the previous actions. The set of actions can be seen in Figure 2.1 as well as the normal flow of messages (failure-free case).

As we mentioned before, one of the proposers has the role of the coordinator and is responsible for starting each phase of the ballot, by executing *Phase1a* and *Phase1b*. The actions *Propose* and *Learn* complete the algorithm, the former is executed by the proposers to propose a value (i.e. submit state-machine command) and the later is executed by the learners to learn the decision of a consensus instance (i.e. assign a command to a log position).

[2]Note that in the context of the pure Paxos protocol the concept of *Learned value* is actually the definition of *decided* given in the classic SMR section (§2.1.2), that is, the value will appear in some position i of the log.

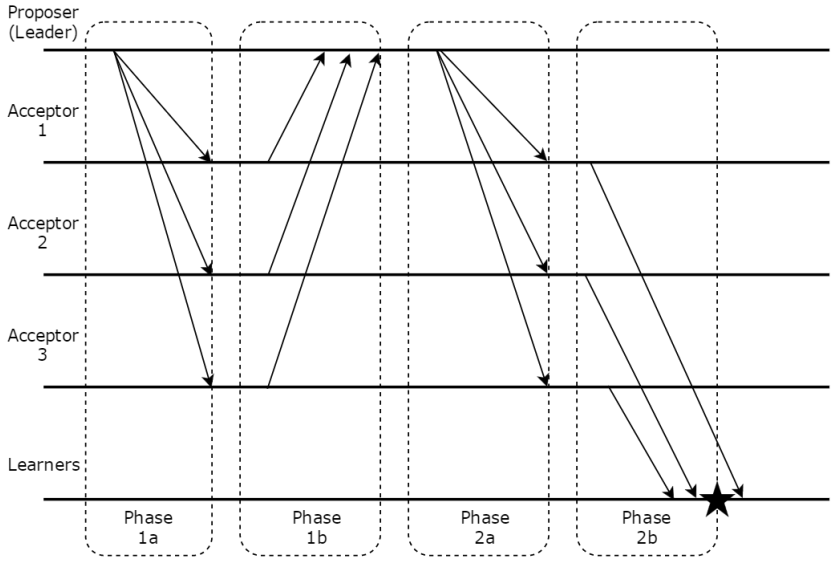


Figure 2.1. Classic Paxos message exchange in the failure-free case. The star marks the learning of a command.

A learner l can only learn the decision of an instance after the execution of the two phases of some ballot m. Furthermore, a learner l can only learn a value that has been proposed by some proposer p. It's important to note that the *Propose* action must happen before the second phase, but not necessarily before the first one.

The algorithm works in the following way, the *prepare* phase is initiated by the leader through the action *Phase1a*, that consists in sending a 1A message to the acceptors, when the acceptors receive this message, they do the following:

- If the acceptor has not responded to any 1A message, he updates its ballot to the ballot of the message 1A (we will call it ballot i) and sends a confirmation 1B message to the leader, through the execution of action *Phase1b*. This way it is guaranteed that the acceptors that replied will not accept values for ballots smaller than i.

- If the acceptor already responded to some 1A message in some ballot s smaller to the current one (i), two scenarios are possible:

  - The acceptor has not yet received any 2A message with a proposal, sent during the coordinator's *accept* phase. In this case, the acceptor updates its ballot to the greatest ballot received (i) and sends a 1B message with this ballot i as parameter to the leader;

– The acceptor has received some $2\mathrm{A}$ message in some ballot k and it must have received a value v proposed by some proposer acting as coordinator in ballot k ($k \leq s < i$), this means that v is the accepted value (its "vote") by this acceptor in the last ballot k in which he accepted (voted for) something. The acceptor then sends back a $1\mathrm{B}$ message to the coordinator of i, containing as parameters (i, k, v), which are respectively the current ballot, the latest ballot k in which the acceptor accepted something and the value accepted in k.

The second phase *accept* is initiated once the coordinator receives messages for the round i from all acceptors in a majority set Q. The coordinator then executes action *Phase2a* sending a $2\mathrm{A}$ message to the acceptors, the message contains as parameter (i, v), which are respectively the current round and the proposal value v selected by the coordinator. The picking of the value is based on these criteria:

- If the coordinator has received one or more $1\mathrm{B}$ messages with values accepted by the acceptors, he then selects the value v of the proposal with the highest ballot number.
- If no $1\mathrm{B}$ message received has any accepted value, the coordinator picks any proposed value to be the proposal.

When an acceptor receives a $2\mathrm{A}$ message with a value v for the round i, he accepts the proposal (i.e. votes for it) if it has not responded to a ballot larger than i. Then, the acceptor through the execution of action *phase2b* sends a $2\mathrm{B}$ message confirming the accepted value v to the set of learner processes. Its important that one understands the distinction between an *accepted* value and a *decided* value; the former only means that the acceptor "voted" for such value, while the latter means that a majority of acceptors voted for the same value in the same round and that the value will eventually be learned[2] by the learners. Finally, if a learner l receives $2\mathrm{B}$ messages from a majority of acceptors containing a value v, then it knows that v was decided and can be safely learned.

In case of failure suspicion (information given by the failure detector oracle), the coordinator initiates a new ballot by executing the *prepare* phase again. Since a coordinator sends the value to be accepted only at the beginning of the second phase (i.e. *accept* phase), the first phase of the algorithm can be executed before receiving any proposal.

The coordinator can execute the *prepare* phase *a priori* for all consensus instances, thus, in the failure-free case a decision takes three message delays in

the critical path (counting the message delay necessary for a client's command to reach a proposer) and $O(n)$ message complexity.

2.1.3 Generic SMR

In their seminal works, Pedone and Schiper [112] and concurrently Lamport [90] introduce an alternative approach to Classic SMR. They make the key observation that if commands submitted to the state machine commute, then there is no need to order them. Leveraging this, they replace the totally-ordered log used in Classic SMR by a partially-ordered one. We call this approach Generic SMR.

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

In Generic SMR, each variable log_p is a partially ordered log, i.e., a directed acyclic graph [90]. In this graph, vertices are commands and any two conflicting commands have a directed edge between them. We use $G.V$ and $G.E$ to denote respectively the vertices of some partially ordered log G and its edges. The append operator is defined as follows: $G \bullet \mathsf{c} := (G.V \cup \{\mathsf{c}\}, G.E \cup \{(\mathsf{d}, \mathsf{c}) : \mathsf{d} \in G.V \wedge \mathsf{d} \asymp \mathsf{c}\}$. A command is decided once it is in the partially ordered log. As previously, it gets executed once all its predecessors are executed.

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

To state this precisely, consider two partially ordered logs G and H. G is prefix of H, written $G \sqsubseteq H$, when G is a subgraph of H and for every edge $(\mathsf{a}, \mathsf{b}) \in H.E$, if $\mathsf{b} \in G.V$ then $(\mathsf{a}, \mathsf{b}) \in G.E$. Given a set $\mathscr{G}$ of partially ordered logs, H is an *upper bound* of $\mathscr{G}$ iff $G \sqsubseteq H$ for every G in $\mathscr{G}$. Two logs G and H are *compatible* iff they have a common upper bound.[3] By extension, a set $\mathscr{G}$ of partially ordered logs is compatible iff its elements are pairwise compatible.

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

[3] In [90], compatibility is defined in terms of least upper bound between two c-structs. For partially ordered logs, the definition provided here is equivalent.

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

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

2.1.4 Leaderless SMR

Classical SMR protocols such as Paxos [88] and Raft [109] rely on a leader replica to order state-machine commands. The leader orchestrates a growing sequence of agreements, or consensus, each defining the next command to apply on the state-machine. As seen in §2.1.3, Generic SMR protocols like Generalized Paxos [90] improve over classic SMR protocols by ordering only non-commuting state-machine commands, which leads to faster decisions once agreement is only necessary for conflicting commands. Nonetheless, as is the case with Classic SMR protocols it still relies on a leader replica to order conflicting commands. The leader based approach has clear limitations, especially in a geo-distributed setting. First, it increases latency for clients that are far away from the leader. Second, as the leader becomes a bottleneck or its network gets slower, system performance decreases. Last, this approach harms availability, because when the leader fails, the whole system cannot serve new requests until an election takes place.

To sidestep the above limitations, a new class of leaderless protocols has recently emerged [104, 106, 16, 62, 131, 57]. These protocols allow any replica to make progress as long as it is able to contact enough of its peers. Mencius [104] pioneered this idea by rotating the ownership of consensus instances. Many other works have followed, and in particular the Egalitarian Paxos (EPaxos) protocol [106], which takes a Generic SMR approach, leveraging commutativity between state-machine commands, but does not rely on a single distinct replica to order conflicting ones.

Although promising, the leaderless approach is not free of problems. We provide evidence for such assertion in §2.4 and dedicate Chapter 3 entirely to explore in-depth this new class of protocols.

Example (EPaxos)

As Generalized Paxos [90], EPaxos orders only non-commuting, aka. conflicting (§2.1.3), state-machine commands. To this end, the protocol maintains at each replica a directed graph that stores the execution constraints between commands. Execution of a command proceeds by linearizing the graph of constraints.

Since any process can be responsible for a command c, it can be the case that each process observes the execution constraints for that command in a different way, this is encapsulated in EPaxos via a set of dependencies of a c, noted: *deps*(c). In EPaxos, processes need to agree on these dependency sets to achieve linearizability [80] (first protocol invariant). To grasp this concept, let us observe the example of a failure-free execution of the protocol in Figure 2.2. In the example, process p_1 is responsible for command c_1 and process p_2 is responsible for command c_2 (both commands conflict with each other). A command c is *committed* once *deps*(c) is known at some process p. Command c can be executed once the transitive closure of its dependencies is committed. To commit the commands the processes p_1 and p_2 proceed in a two message delay communication pattern (sending PREACCEPT messages and receiving PREACCEPTACK) where they report each others dependencies for their commands (initially $\varnothing$ for both, as they have seen no conflicting commands prior) and contact enough peers to guarantee that at the end of this communication the second invariant that guarantees linearizability will hold, that is, that one of the two will hold: $c_1 \in deps(c_2)$ or $c_2 \in deps(c_1)$ at all processes (i.e. conflicting commands will be ordered). We can see in the example that this is the case at p_1 where the process is able to commit for c_1 with $deps(c_1) = \varnothing$ and p_5 commits c_2 with $deps(c_2) = \{c_1\}$.

Note that the dependencies reported by p_2 and p_3 were equal, that is what allows p_1 to commit earlier (this is called a *fast path*, that is, spontaneous agreement among processes). This is not the case at process p_5 (dependencies reported by p_3 did not match the ones reported by p_4), forcing p_5 to go through an agreement[4] phase, a two message delay communication pattern (sending ACCEPT messages and receiving ACCEPTACK messages) where at the end, processes agree on the dependencies of c_2.

2.2 Byzantine Fault Tolerance

Replicated services are becoming increasingly more geo-distributed and open (no longer operated by a single entity), this leads to systems being ran in a wider number of platforms and programmings languages, resulting in more harmful bugs. Coupled with this, we have the growing reliance of public, industry and government on such systems, which makes them profitable targets for exploitation by different parties. The combination of these factors require that leveraging simple benign failures is no longer enough to build services that behave correctly.

[4]Hence, in the common case, EPaxos executes a command after two message delays if the fast path was taken and four message delays otherwise.

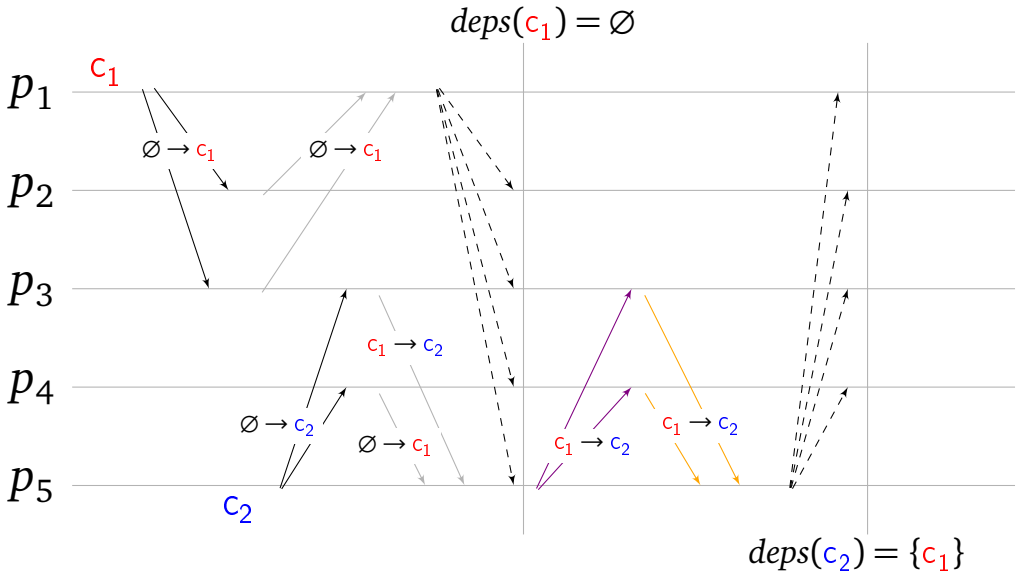


Figure 2.2. EPaxos normal-case. $N = 5, f = 2$. Messages and their corresponding phases are described by the following colors: **PREACCEPT**, PREACCEPTACK, ACCEPT, ACCEPTACK, *COMMIT* (dashed lines).

A more fitting failure assumption is captured under the term *Byzantine failure* [93], where processes might deviate arbitrarily from their assigned specification or rationally misbehave in a way that causes the most damage to the system. A Byzantine Fault-Tolerant (BFT) service is therefore a system that strives to give correct answers to its clients despite the presence of Byzantine processes.

2.2.1 System Model

To account for Byzantine failures we consider a different system model than §2.1.1. This new model is an adaptation of the one found in [41]. Further, readers should be aware that the definitions found in §2.1.1 are often overwritten.

We consider the standard model of wait-free computation in a partially synchronous [58] distributed system consisting of the fixed set of processes $\Pi = \{p_1, p_2, \ldots, p_n\}$, with $n \geq 3f + 1$ processes where f is the maximum number of Byzantine failures tolerated by the system. When a process is Byzantine we call them *faulty* or *byzantine*, otherwise we call them *correct* (we also consider that a correct process never crashes).

Further, we assume that processes communicate via message exchange over

authenticated and reliable channels and the existence of collision-resistant hash functions, digital signatures and a public-key infrastructure (PKI). Any correct replica, can authenticate messages it sends by signing them. We note that a message m is signed by x as $\langle m \rangle_x$ We also assume that faulty processes are computationally bound so that whp. they are unable to break the cryptographic techniques mentioned. It follows that state-machine commands are non-craftable as they are considered to be signed correctly.

2.2.2 Byzantine Fault-Tolerant State-Machine Replication (BFT SMR)

We define Byzantine Fault-Tolerant State-Machine Replication (BFT SMR) by adapting the definition in §2.1.2 to account for Byzantine failures. As common in the literature [41], this means we consider that the safety properties must hold at correct processes. Differently than §2.1.2 (but similar to [139]), we drop the proxy model and consider that commands are submitted by clients and we don't consider them to be byzantine. The modified properties of the service are the following:

Validity: A command is decided once and only if it was submitted by a correct process before.

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

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

2.2.3 Key Notions

In this section we introduce a series of key notions that have seen pervasive use in both modern and classic BFT protocols.

Byzantine Quorum System

Under the fail-stop model, quorum systems are a well-known tool to improve availability and efficiency of replicated data [8, 67, 78, 1]. Essentially, a quorum system is a collection of subset of processes (i.e *quorums*) from the universe of processes, such that each pair of these sets have non-empty intersection. Quorums can run independently (i.e each quorum can make progress at its own

speed), thus increasing the system's performance and availability while safety properties hold by relying on the intersection property.

Byzantine quorum systems extend the classical quorum systems to cover byzantine failures. The definitions that follow are taken from [101]: a *quorum system* $\mathcal{Q} \subseteq 2^{\Pi}$ is a non-empty set of subsets of Π, every pair of which intersect. Each $Q \in \mathcal{Q}$ is called a *quorum*. A *fail-prone system* $\mathcal{B} \subseteq 2^{\Pi}$ is a non-empty set of subsets of Π such that none of its elements is contained in another and that some $B \in \mathcal{B}$ contains all the *faulty* nodes. The fail-prone system characterizes all the failure scenarios possible in any execution of the system. A *Byzantine Quorum System* (BQS) is a pair of a quorum system and a fail-prone system: $(\mathcal{Q}, \mathcal{B})$.

The simplest BQS is the *masking quorum system*. A masking quorum system satisfies the following properties:

M-Consistency $\forall Q_1, Q_2 \in \mathcal{Q}. \forall B_1, B_2 \in \mathcal{B} : (Q_1 \cap Q_2) \setminus B_1 \not\subseteq B_2$

M-Availability $\forall B \in \mathcal{B}. \exists Q \in \mathcal{Q} : B \cap \mathcal{Q} = \varnothing$

Informally, M-Consistency guarantees that any two quorums intersect in $2f + 1$ processes and therefore in $f + 1$ correct process. M-Availability ensures the existence of a quorum containing only correct processes.

A masking quorum system can be used to mask the arbitrarily faulty behavior of data repositories. This means that, for instance, it can be used to build a service that offers read and write operations to a replicated variable x and guarantees safe variable semantics [13] despite the presence of B faulty replicas. From the example in [101], consider that Q_1 is the quorum used to write to variable x and Q_2 the one used to read it, then a client that uses Q_2 observes the following: the correct value for x is obtained from each replica $(Q_1 \cap Q_2) \setminus B$, possible by M-Consistency.

Now, if we want to implement a service that will be used as a repository for information that only clients can produce and to which they can detect attempts of modification by a faulty process (e.g. clients signing messages via digital signatures), the cost of accessing the replicated data is decreased (i.e. the quorums intersection size decrease by a factor of f). The *dissemination quorum system*, captures such notions and is defined through the following properties:

D-Consistency $\forall Q_1, Q_2 \in \mathcal{Q}. \forall B \in \mathcal{B} : Q_1 \cap Q_2 \not\subseteq B$

D-Availability $\forall B \in \mathcal{B}. \exists Q \in \mathcal{Q} : B \cap \mathcal{Q} = \varnothing$

Informally, D-Consistency ensures that any two quorums must intersect in a correct process. D-Availability ensures the existence of a quorum containing only correct processes.

To measure the performance of a quorum system we borrow the definition of load from [101]. Load of a quorum system can be informally defined as the minimal access probability of the busiest server in the system, minimizing over all strategies for picking quorums.

Threshold Signatures

A classical digital signature scheme [76] consists of a collection of algorithms: key generation, signing and verification. The input for key generation is a security parameter and the output is a pair of public key and private key. The input for signing is a private-key and a message m and the output is a signature σ.

The idea of a threshold signature scheme was introduced in [51, 52] with the intent of distributing trust among parties in the production of digital signatures. A (k, n)-threshold digital signature scheme consists of a protocol for n processes from which k can cooperate to generate valid signatures. The scheme works by dividing a private key through the n processes such that any set of k process can contribute a partial signature to a message m. The partial signatures are then sent to a combining process, which combines the partial signatures into a valid threshold signature on m. The scheme guarantees whp. that any set of processes with size smaller than k is unable to generate a valid threshold signature on m.

This type of schema has been present in a vast number of modern BFT algorithms and systems [10, 139, 38, 74]. Such a non-interactive scheme is used in protocol like HotStuff [139] and SBFT [74] to allow a process to justify its vote for a specific value by using a single threshold signature generated from k partial signatures, which saves a factor of k in terms of communication complexity.

Verifiable Random Functions (VRFs)

Verifiable Random Functions [105] are pseudo-random functions [75] typically defined through a collection of polynomial-time algorithms and a set of requirements (formally defined in [105]).

Adapted from [25], we say that $VRF = (\mathrm{VRF}.Gen, \mathrm{VRF}.Eval, \mathrm{VRF}.P, \mathrm{VRF}.V)$ consists of the following:

- a probabilistic key sampler $\mathrm{VRF}.Gen(seed)$, that takes a $seed$ as input and outputs a secret key SK and public verification key VK
- an evaluator $\mathrm{VRF}.Eval_{SK}(x)$ that takes as a parameter the secret key SK and x and outputs y

- a prover $\mathrm{VRF}.P_{SK}(x)$ that takes as a parameter the secret key SK and outputs a proof π that y is consistent with the verification key VK

- a verifier $\mathrm{VRF}.V_{VK}(\pi, x, y)$ that verifies the proof.

Informally, it is possible to verify that an output y from a VRF corresponds to a correct evaluation of the function on any given input x. The secret key allows anyone to compute the function $y = \mathrm{VRF}.Eval_{SK}(x)$ at any point x, and also to compute a proof $\pi_{x,y}$ that y was computed correctly. VRFs requirements guarantee that an adversary has negligible probability to distinguish the output y of the function at any point x, from a pseudo-random value.

VRF is a type of primitive that has seen recent use in a variety of BFT protocols [70, 95], it allows for example to select in a pseudo-random, verifiable and non-interactive manner (i.e. no message exchange) a subset of processes responsible for important protocol roles (e.g. a proposer).

2.2.4 Historical Solutions

The term *Byzantine failures* is presented in [93] where the authors formulate the problem of reaching consensus under these types of failures through the Byzantine Generals Problem. A synchronous solution can be found in [55], an improvement over the first solution for the problem found in [111]. This improved version has $O(n^3)$ communication complexity (optimal as seen in [54]). A leader driven synchronous protocol using randomness is presented in [84] with expected constant-round solution and resilience of $(n-1)/2$.

Considering an asynchronous model we know by [64] that there is no deterministic solution for consensus in the possibility of a single failure. A circumvention of the result can be found in [21] where randomness is used for processes to eventually converge to consensus (through independent random coin flips). These ideas are enhanced in [38] where the authors present a protocol that uses cryptographic methods to share an unpredictable coin, this allows for a total communication complexity of $O(n^3)$.

Another approach to circumvent the impossibility result is to consider a partially synchronous model. The first work to go in this direction can be found in [58], where the authors present a protocol that remains safe while the system behaves asynchronously and guarantees termination after the system reaches synchrony. During synchronous periods the protocol displays a communication complexity of $O(n^4)$ and $O(n)$ rounds per agreement.

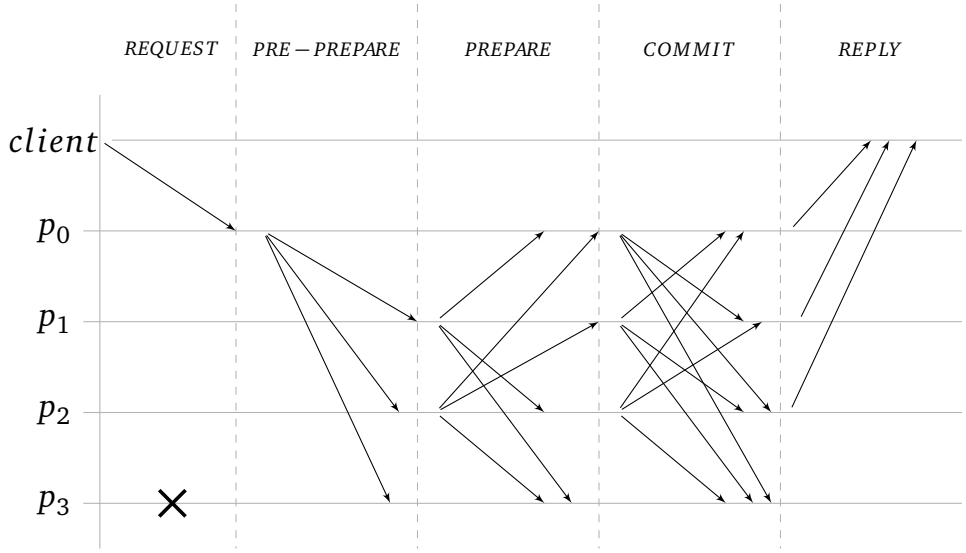


Figure 2.3. PBFT phases in the normal-case.

2.2.5 Practical Byzantine Fault-Tolerance (PBFT)

In the BFT SMR realm of protocols, PBFT [41] is the baseline work for all protocols that came after it. It presents a very pragmatic way of doing Byzantine fault-tolerance replication, in the sense that it does not rely on synchrony assumptions for safety, only liveness and it does not rely on randomization either. Assuming that up to f replicas can be Byzantine, PBFT optimally requires $3f+1$ replicas to ensure safety and progress.

It achieves this through an algorithm that combines a primary-backup schema and quorum replication techniques [69] to order clients commands and optimizations, such as the use of symmetric cryptography to authenticate messages. Processes take part in a series of *views* (i.e. configurations) each coordinated by a leader that after 2 round-trips orders a command. In PBFT, quorums are simply defined as sets of $2f+1$ processes, nonetheless they still abide to the properties of a dissemination quorum system (§2.2.3), hence it enables the use of quorums as a reliable memory for protocol information. The information is written to quorums and replicas collect *quorum certificates*, which are sets with one message from each quorum participant saying that it logged the information.

In the normal-case the protocol roughly works as follows: Once the primary (i.e. leader) receives a client's request m, it attaches a sequence number n to it and broadcasts a PRE-PREPARE message to all replicas. This message also contains the view v in which the message is being set. A replica will only accept the message if it is in the same view v, if it can verify the message's authenticity,

if the sequence number n is valid (to prevent a byzantine leader exhausting the space for sequence numbers by picking a very large one) and if it hasn't accepted a PRE-PREPARE message for v and sequence number n containing a different client request.

Once the PRE-PREPARE message is accepted the replicas broadcast to all replicas a PREPARE message (this messages indicates that the replicas have agreed to assign n to m in view v). Once a replica collected $2f + 1$ matching PREPARE messages for sequence number n, view v and request m we say that a *prepared certificate* is done. It indicates that replicas have agreed on an order for requests that are in the same view.

Now the protocol proceeds with an extra phase to guarantee a total order for requests across views, as it could have been the case that a view change occurred and replicas could have collected prepared certificates in another view with the same n and different m. Replicas broadcast to all others a COMMIT message and again collect $2f + 1$ of these messages (to form a *commit certificate*), by which afterwards we can say that the request is *committed*.

Each replica then executes the requested operation m (if it has executed all requests with lower sequence number than n) and replies to the client. Once a client receives $f + 1$ matching replies it commits the request.

In terms of performance, when the leader of a view is correct, the message complexity of the protocol is of $O(n^2)$, when a *view-change* (i.e. reconfiguration) occurs, the message complexity is of $O(n^3)$.

Understanding the $3f + 1$ bound

To understand why PBFT optimally requires $3f + 1$ replicas to ensure safety and liveness let us start by looking at an example (based on [123]) that does not consider byzantine failures (modeled in §2.1.1).

Consider a non-Byzantine replicated service consisting of n replicas from which f may fail-stop and that maintains mutual exclusion on *Read (R)* and *Write (W)* operations to a mutable variable. Let us consider that such service is implemented via Paxos, this means that only $2f + 1$ replicas are required for safety and liveness [88]. To guarantee liveness the service may have to return a reply to an operation before the operation is received by all replicas, since f replicas might not respond. So if we consider that an operation W is done, the client that issued the operation must wait for no more than $n - f$ replies. Now consider that a read operation followed, the client waits for $n - f$ replies, but this time the replies might come from another set of replicas. Thus, the number of replicas in the intersection used for the operations W and R is of $n - 2f$. So that

safety holds (e.g. the operation R doesn't miss W), there must be at least one non-faulty replica in such intersection (i.e. $n-2f > 0$), thus at least $2f+1$ replicas are necessary in order for safety and liveness to hold. An attentive reader might recall that this ties back to the Paxos explanation given in §2.1.2 that a necessary condition to decide a proposal in an instance of the protocol was that a reachable *majority* (i.e. a *quorum* of size $f+1$) of acceptors exists.

Now if we try to cast such a service to tolerate Byzantine failures using the previous $n = 2f+1$ bound, we will reach the following problem: We cannot distinguish if the f unresponsive replicas that did not reply for the operations R and W are slow or Byzantine. If we consider that they were slow, then it might be the case that f Byzantine replicas are still present in the replicas used to reply to the operations R or W. It follows that we can have f Byzantine replicas among the $n-2f$ ones, choosing arbitrarily to which operation to reply and to which not to reply. To fix this issue the intersection needs to be increased by a factor of f (i.e. $n-2f > f$), leading us to $n > 3f$ and thus at least $3f+1$ replicas are required in the Byzantine case to guarantee safety and liveness if we assume that up to f are Byzantine.

2.2.6 700BFT

PBFT is notably a complex protocol, properly implementing the view-change mechanism [4] and proving its correctness is a non-trivial task. Recent works have been taking the direction of exploring the modularization and composability in BFT SMR, for less complex designs and/or performance gain [17, 33, 136]

In [17], the authors decompose BFT SMR in a series of composible modules. One of these modules is Quorum, a client-based BFT protocol that allows for the abortion of commands. It achieves the optimal minimum latency in contention-free cases by using a simple message pattern: clients broadcasts their requests to $3f+1$ replicas, each replica executes the request and replies back to the clients. If all the replies match, the clients completes; otherwise it aborts and another BFT protocol is launched (e.g. PBFT).

2.2.7 Query/Update (Q/U) and Hybrid Quorum (HQ)

In [2] the authors present Q/U a byzantine fault-tolerant protocol that is quorum based. This means that differently than agreement based protocols like PBFT where replicas are required to process all requests, in a quorum based approach this is not the case. This allows Q/U to have better fault scalability and latency when compared to the agreement approach of likes of PBFT, since a client can

communicate exclusively to a quorum. A trade-off can be observed though, once Q/U requires $5f+1$ replicas to tolerate f byzantine ones, not optimal as seen in §2.2.5. The Q/U protocol provides an interface based on the operations: *query* that do not modify the replicated objects and *updates* that do. Such interface allows byzantine-fault tolerant services to be built on top of it similarly to BFT SMR. Q/U works in a simple message pattern, where a client sends operations exclusively to a preferred quorum of size $4f+1$ and the quorum replies back to the client. Consistency on the replicated objects hold without the need of a leader process because the protocol maintain versions of the objects. Such versions are created once an update operation is invoked and the history of the most up-to-date object versions is exchanged among replicas and clients, that is, processes return histories to clients in response to requests to guarantee that the last version of an object is being queried.

Since a client can communicate exclusively to a preferred quorum, it might be the case that some replicas have an outdated history of an object, which requires that outdated replicas query $f+1$ other replicas for the most up-to-date history of an object. If a client detects that different versions of an object exist in different replicas (concurrent updates may lead to different object versions), it initiates a *repair* phase to bring outdated copies of an object to the most up-to-date version. Such phase is expensive and highlights the problems of the protocol to deal with contention.

The protocol Hybrid Quorum (HQ) [45] works similarly to Q/U but uses a protocol similar to PBFT to resolve the contention issue and requires optimally $n = 3f+1$. In the case that no contention occurs, HQ relies on a byzantine quorum system where quorums have size of $2f+1$. It allows for write operations to take two round trips between clients and replicas and reads to take only one. When contention between operations occur it orders them via a variation of PBFT.

2.3 Blockchain

Blockchain has been the focus of a lot of hype and news since its inception in 2008 with Bitcoin [107]. Even though a lot of research [14, 110, 66, 11] exists on the theme, scientific consensus hasn't been reached on a single formal abstraction of these objects. Thus, let us consider first an informal definition based on the one found in [14], where we can say that a blockchain is a replicated tamper-proof append-only ledger, that is, an indelible log of transactions that takes place between various entities, where transactions are stored in blocks and each block

is cryptographically linked to a previous one.

The first critical idea for making sense of the blockchain landscape lie in understanding how a blockchain deals with *membership management*. Initially all blockchains were *permissionless* (i.e public), where the parties participating in the protocol cannot be identified reliably and anyone can join the network. Recently there has been the rise of a new wave of protocols aimed at providing the same abstraction but under an environment that resembles more the classical BFT system model, these are *permissioned* (i.e. private) blockchains, where the parties taking place in the protocol have reliable identities and have been vetted to participate.

The second critical idea to make sense of blockchains lie in understanding the *consistency guarantees* provided by them. Ideally a replication scheme is implemented by keeping all replicas synchronized after each update to the shared object. This ideal model is called *strong consistency*, or linearizability [80]. Typically, permissionless blockchains offer some kind of weaker consistency guarantee whereas in permissioned ones linearizability is the ruling consistency criteria.

Understanding the 'why' behind this relation allows one to grasp the actual difficulty of the problem at hand. We explore this idea more in-depth in the following section where we present the different types of blockchains and their protocols.

Permissionless Protocols

Perhaps the best way to understand the abstraction provided with a blockchain consists in understanding how it differs (a similar approach is taken in [83]) from the classic BFT SMR abstraction presented in §2.2.2. Permissionless blockchains distinguish themselves primarily from classical BFT SMR by having an open membership, this affects the difficulty of doing replication through classical means. The reason for this is that behind any SMR protocol is a *consensus* protocol, this is also the case for blockchains. In the context of blockchains, processes propose transactions to be added to the ledger and from these proposals one is chosen.

As is the case with classic BFT SMR, processes repeatedly run a consensus protocol, in this case to append a batch of transactions to the ledger. If one just runs a byzantine fault-tolerant consensus algorithm as PBFT, out-of-the box in an open membership model, its system would be vulnerable to a wide-range of attacks, as for example: *Sybil* [56] attacks, that is, with unreliable identities an attacker can flood the consensus protocols with proposals made by puppet processes controlled exclusively by himself. In [18] the authors prove that such type of attack is fundamentally related to the permissionless model.

Viewing it from another angle, in classical BFT SMR protocols, consensus is reached once a *quorum* of participants agree on the same decision, the size of a quorum and thus their intersection size is typically based on the amount of failures the system is supposed to support, therefore in an open membership model is harder to guarantee what size of quorum is enough to guarantee that safety properties will not break.

To deal with open membership and the sybil attack mentioned above, permissionless blockchains often abstract what a failure means in classic BFT to a different adversary model [110] where the adversary is typically bounded by computational power or economic means.

Although it is out of the scope of this thesis to formalize blockchains, in the interest of the reader we provide a sketch (based on the works in [110, 83, 11]) of how to redefine BFT SMR (§2.2.2) to permissionless blockchains. Consider that each log position holds a batch of transactions (i.e. commands) let us call it a *block*. Clients submit blocks to be appended to the *log* but only blocks that satisfy some validity predicate P can be appended. Predicate P is dependent on the application and it abstracts the creation process of a block. An example of definition of such predicate could be the following: A block abides to predicate P iff it can be connected to a previous block currently in the *log* and it does not contain duplicate transactions. Differently than classic BFT SMR, in permissionless blockchains typically safety properties require that correct processes agree only on a prefix (that exists whp.) of the *log*. It allows for the existence of a suffix of the *log* of size T consisting of "unconfirmed" blocks. The properties required to implement a permissionless blockchain are then the following:

Validity: A block is decided once and only if it satisfies predicate P.

Stability: For any correct process p, at any given time log_p is prefix of itself at any later time, whp. and when ignoring a suffix of size T.

Consistency For any two correct process p and q, at a certain time and whp. log_p and log_q are compatible when ignoring a suffix of size T.

2.3.1 Bitcoin

One of the primary goals of Bitcoin [107] was to implement a decentralized currency, that is, a currency that wouldn't depend on a single entity (like a bank) or be vulnerable to censorship by any government, very much in the vein of the the cypherpunk manifesto [48]. The solution found by the anonymous author

(or authors) Nakamoto, to implement such large scale geo-replicated service relies on integrating Proof-of-Work (PoW) [59] (a technique originally created for spam control) to a large-scale consensus protocol. Such idea spawned the entire blockchain field by providing a mitigation mechanism to the aforementioned Sybil attack.

In more detail the Bitcoin protocol works as follows: Processes communicate via reliable FIFO authenticated channels (implemented with TCP), thus modeling a partially synchronous system [58]. When a transaction is sent by a client it is placed in a shared pool, processes called *miners* collectively run a repeated consensus protocol to select which transaction will be appended to the ledger. In this consensus protocol, miners solve a cryptographic puzzle (proof-of-work) to produce a valid block of transactions (i.e. miners are basically going through a voting process, where they vote with their CPU power for valid blocks). This proof-of-work procedure is computationally expensive (therefore, economically expensive as well) to curb the effectiveness of Sybil attacks.

The advantage of using this approach is that it scales well with the numbers of miners if one consider the system's safety, since the more miners there are, the more secure the service becomes (i.e. it gets increasingly more expensive to do attacks). The downside is that Nakamoto's consensus it is not strictly speaking consensus. It can be the case that two miners concurrently solve the puzzle and then append blocks to the blockchain in a way that neither block precedes the other. This is called in the Bitcoin community as a *fork*. Typically, processes continue to build on the block with the longest chain, that is, the one which has more work done.

Dealing with forks highlight the disadvantages of the protocol, for instance: it takes a waiting time of about 10 minutes to grow the chain by one block and from an application point of view, waiting a few blocks is required (6 are recommended in Bitcoin [26]) to guarantee that the transaction remains in the authoritative chain. Therefore, the possibility of forks affects the consistency guarantees and throughput of transactions (about 3 to 10 transactions per second in Bitcoin).

Many papers [11, 66, 110, 73] have been published with the intent of formalizing PoW protocols like Bitcoin and studying its consistency guarantees. In [66] the authors provide one of the first formalizations of the Bitcoin protocol and show that under synchronous environment assumptions the protocol guarantees whp. an eventual consistent prefix. More broadly, in [11] the authors introduce the notion of *Eventual Prefix* and show that Bitcoin and other permissionless protocols abide to it. In [73, 72] the authors argue that Bitcoin implements *Monotonic Prefix Consistency* whp.

Consistency issues aside, abandoning PoW is a necessity since Bitcoin alone

was estimated to consume 59TWh [27] in 2019. Such fact represents why the PoW approach is not scalable/sustainable and why the most recent permissionless protocols do not follow such approach.

2.3.2 Algorand

To address the problems of power consumption and low-throughput that PoW based consensus protocols face, new types of consensus protocols [36] aimed at the permissionless model have been under development that rely on specific cryptographic methods and assumptions.

This is the case with Algorand [70], where the combination of Verifiable Random Functions [105, 25](VRFs) and a new mechanism called Proof-of-Stake (PoS) [22] allows to alleviate some issues with PoW consensus. In a PoS schema, the voting power of processes is proportional to their money (i.e. stake) in the system and not their computational power. Nonetheless, such mechanism brings its own shortcomings [135].

The Algorand protocol works in *rounds* where each round has two phases. In the first phase processes elect a leader via cryptographic sortition, this leader is responsible for proposing a new block to be appended to the ledger. During the second phase, a subset of processes is picked in a pseudo-random fashion to become part of a *committee* that will run a modified version of PBFT [41]. The cryptographic sortition mechanism guarantees that leader and committee members are elected proportionally to their *weights* (i.e. money in the system or stake), therefore the higher their weights the higher the odds to propose a block (as a leader) and as committee member to be able to vote and influence the final decision.

Algorand allows for faster confirmation times and lower energy consumption when compared to PoW protocols, nonetheless the consistency guarantees provided are still probabilistic [11]. Scalability wise, the protocol supports thousands of nodes but in a real world scenario it makes distinction between them, notably relay nodes that by default hold the entire history of the ledger and are primarily used to route the network traffic to a set of non-relay nodes. In [94] the authors mention that although both types of nodes participate in consensus, Algorand recommends that only non-relay nodes participate in consensus. Interestingly enough in the Algorand FAQ [28] is possible to note that the control over relay nodes is highly centralized and as of January 2021, there is just over 100 relay nodes.

2.3.3 Stellar

Differently from the Bitcoin and its PoW consensus and Algorand with its consensus powered by VRFs and PoS, Stellar [98] takes a more classic approach by relying on a special case of byzantine quorum system modified to support open membership. Stellar's assumptions are specified using a Federated Byzantine Quorum System (FBQS) [68], where each participant in the Stellar Consensus Protocol (SCP) declares its own set of processes that is trusting on.

The original Stellar whitepaper has no mentions of former work on Byzantine quorum systems. The relation between SCP, Bracha's Byzantine reliable broadcast [32] and Byzantine quorum systems was only studied in-depth in [68], where the authors try to shine a light into SCP by showing that the intricate federated voting procedure of the protocol is equivalent to using Bracha's Byzantine reliable broadcast in a dissemination quorum system. The lack of formalization of the system as a whole is evidence that the task of safely porting classic approaches to an open membership setting is challenging and a recurring issue in the realm of new permissionless blockchains.

Permissioned Protocols

The main idea behind permissioned blockchains is that the application is run by reasonably trustworthy parties (e.g. companies) that don't fully trust each other and that desire that no specific party completely takes over the process of adding new blocks to the blockchain. Under the permissioned setting, processes identities are known and membership control is strict (e.g. via accountability [79, 114]), thus the model assumptions favors the use of classical BFT protocols, that differently than PoW or protocols from the permissionless setting, generally provide stronger consistency guarantees and better throughput.

Nonetheless, permissioned blockchains envision a scale that surpasses the expectations of classical protocols (a typical deployment would be five or seven replicas for PBFT). A quadratic message complexity and/or a leader-driven approach are basically prohibitive when considering hundreds or thousands of replicas. Hence, permissioned protocols are essentially BFT SMR protocols optimized to deal with such problems.

In what follows, we describe some modern BFT SMR protocols that aim primarily at blockchains, highlighting their core mechanism, advantages, disadvantages and message complexity (fundamental to analyze a protocols scalability).

2.3.4 HotStuff

HotStuff [139] builds upon PBFT two-phase core but adds an extra protocol phase (PRE-COMMIT) to each view and relies on threshold signature scheme to generate a representation of $(n-f)$ signed votes (i.e. quorum certificate) as a single signature (PBFT's generate similar certificates but via an all-to-all communication). In detail this is achieved by a protocol that reaches consensus after four phases, where each phase is a back and forth between the leader and the replicas.

The protocol starts with a PREPARE phase, where the leader initiates a view by broadcasting a value to all processes and collecting $2f+1$ signatures on this value (PREPARE certificate), this indicates to the leader that replicas have validated and accepted the value it has proposed. The PREPARE phase is followed by the PRE-COMMIT phase, where the leader broadcasts the PREPARE certificate and collects signatures from the processes, if the leader is able to collect $2f+1$ votes it generates a PRE-COMMIT certificate, it indicates that even if a view-change (i.e. reconfiguration) occurs the value will not change. Followed by the COMMIT phase, the leader broadcasts the PRE-COMMIT certificate and collects enough votes to be able to generate a COMMIT certificate, if this is the case the value is considered decided. In the last phase (DECIDE) the leader broadcasts the COMMIT certificate and processes verify it and decide accordingly.

HotStuff has $O(n)$ message complexity (during view-changes as well). By choosing to trade-off latency for scalability it is able to avoid using a synchronous core (i.e. a leader doesn't need to wait for a fixed delay in order to propose new values), which is the case with Tendermint [34] and Casper [36].

As is the case with PBFT and other modern BFT SMR protocols, HotStuff is leader-based which in this case poses extra problems. Besides being responsible for ordering all commands a leader is also responsible for generating quorum certificates (by combining threshold signatures), which poses availability and scalability issues for modern permissioned blockchains.

2.3.5 SBFT

SBFT [74] (Scalable Decentralized Trust Infrastructure for Blockchains) as HotStuff, lowers the message complexity caused by the all-to-all communication (typical of PBFT-like protocols) by relying on combined signatures generated via threshold cryptography as well as lowering the message complexity in case of view-changes. This is achieved by sharing the burden of collecting and combining threshold signatures by assigning this task to processes that have different

roles in the protocol (commit collectors and/or execution collectors).

The protocol works as follows in the fast path (default mode of execution): After receiving clients requests, a leader creates a decision block with the requests and broadcasts it to the processes (PRE-PREPARE phase). When replicas receive such a message they partially sign the decision block using their threshold signature and send it to the set of collectors ($c+1$ non-leader processes are selected for this task for each view), this corresponds to the SIGN-SHARE phase. Each collector combines the partial signatures and creates a *full-commit-proof* that is sent back to the processes (FULL-COMMIT-PROOF phase), this certificate is enough for the processes to commit. Afterwards the execution protocol begins, this protocol is divided in two phases: SIGN-STATE and EXECUTE-PROOF. In the SIGN-STATE processes sign blocks using $f+1$ threshold signatures and send it to a set of execution collectors. In the EXECUTE-PROOF phase the execution collectors generate a succinct execution certificate and send it to the other processes and to the clients (it indicates to these that the requests were executed). When the fast path cannot progress, a variation of PBFT using threshold signatures is used as fallback.

The protocol has $O(n)$ message complexity in the failure-free case and differently from HotStuff it has a complex view-change mechanism with a message complexity of $O(n^2)$. Sharing the burden of the tasks among multiple processes is an advantage of the protocol but poses its own problems given that a small set of collectors might not provide resilience enough in scenarios where an adaptative opponent can target them.

2.3.6 Pompe

Pompe [140] tackles the problem of adapting classic BFT SMR protocols to their use in blockchains under different optics. Its main concern is not necessarily scalability but to diminish the influence of byzantine nodes on the order of transactions recorded to the ledger. The authors argue as being of key interest in the context of permissioned blockchains the issue that malicious processes can try to push for ordering of commands that financially benefits them. The proposed solution is defined via an architecture that decouples ordering from consensus.

The protocol shares the same model assumptions made in PBFT (i.e. $n \geq 3f+1$, access to PKI, etc) , requires additionally that processes have the ability to produce monotonically increasing timestamps and uses the ideas presented in [19].

The protocol follows a two-phase structure and orders commands based on timestamps. In the first (ordering) phase, a process broadcasts a client's com-

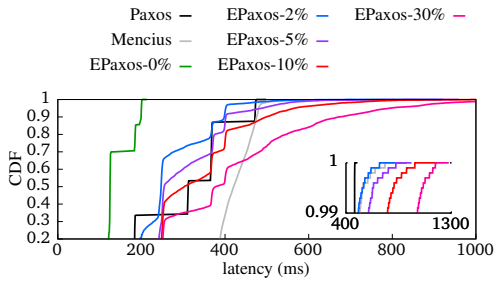
mand wrapped in a signed request message and waits for the response of $2f + 1$ processes reporting their own timestamps for that command. Afterwards, the process wraps the $2f + 1$ timestamps received in a message and broadcasts it, a receiving process calculates the median timestamp from the ones received and accepts the command. If a command is accepted by a quorum of $2f + 1$ nodes, it is guaranteed to be included in the totally-ordered logs of correct processes and its position is determined by the median timestamp (the command is considered *sequenced*). The sequenced commands are not yet suitable for execution, because it must be guaranteed that commands with lower timestamps will not be sequenced. Thus, the protocol proceeds to the second (consensus) phase. This phase takes two message delays for preparation of the proposal and considers that protocols like HotStuff [139] or SBFT [74] can be used as a black-box service to reach agreement.

In Pompe a command takes takes two round-trips to be committed and an extra one round-trip *before* being able to run the consensus service. It is important to notice that all processes take part in the consensus service, further limiting scalability. Not considering the consensus service Pompe has an $O(n)$ message complexity.

2.4 Motivation

One may observe that most protocols described in this chapter rely on a leader replica to establish a total order between client's commands. This approach is known to lead to availability and performance issues that are amplified when considering geo-replicated services. The new class of leaderless protocols introduced in §2.1.4 presents itself as a promising direction to address the aforementioned problem. Unfortunately, this class of protocols hasn't been fully formalized, the protocols in [104, 106, 16, 62, 131, 57] are complex (e.g. EPaxos recovery procedure is notably complex) and each present an ad-hoc approach to leaderless SMR.

The heterogeneity of the protocols from this class leads to a phenomena that can be observed in practical experiments and that up to this point, remained unexplained. To exemplify consider EPaxos, in the common case, the protocol executes a command after two message delays if the fast path was taken, that is, if the replicas spontaneously agree on the constraints of a command, and four message delays otherwise. Unfortunately the latency of EPaxos may raise in practice well above four message delays. To illustrate this point, we ran an experimental evaluation of EPaxos, Paxos [88] and Mencius (one of the first



(a) Latency distribution when varying the conflict rate.

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

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

Figure 2.4. Performance comparison of EPaxos, Paxos and Mencius – 5 sites: South Carolina (SC), Finland (FI), Canada (QC), Australia (AU), Taiwan (TW, leader); 128 clients per site; no-op service.

leaderless protocols) [104] in Google Cloud Platform. The results are reported in Figure 2.4a, where we plot the cumulative distribution function (CDF) of the command latency for each protocol. In this experiment, the system spans five geographical locations distributed around the globe, and each site hosts 128 clients that execute *no-op* commands in closed-loop. Figure 2.4b indicates the distance between any two sites. The conflict rate among commands varies from 0% to 30%.[5] We measure the latency from the submission of a command to its execution (at steady state). Two observations can be formulated at the light of the results in Figure 2.4. First, the tail of the latency distribution in EPaxos is larger than for the two other protocols and it increases with the conflict rate. Second, despite Mencius clearly offering a lower median latency, it does not exhibit such a problem. For a wider adoption of this class of protocols, it is necessary that the root cause of such behaviors to be fully comprehended.

In the realm of Byzantine replication, Leaderless SMR is of particular interest in the context of permissioned blockchains, as one can observe that most BFT SMR protocols presented follow a leader driven approach. Since a Byzantine leader can have more drastic effects in the performance of these applications (i.e. view-change must be triggered and often has quadratic message complexity as seen in [41]), modern BFT SMR protocols like HotStuff [139] and SBFT [74] leverage the fact that leader change might happen often and focus on diminishing the cost of it.

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

Platform	Membership	Consensus Mechanism	Linearizability	Throughput	Scalability
Bitcoin [107]	Permissionless	PoW (Proof-of-Work)	No	3-10 tx/sec	∼11k nodes **
Algorand [70]	Permissionless	PoS (Proof-of-Stake)	No	∼100 tx/sec	10k nodes
HotStuff [3]	Permissioned	BFT SMR	Yes	∼15k tx/sec	128 nodes*

Table 2.1. Performance of blockchains. *: Scalability for inter-replica latency 10ms +-1.0ms, with 0/0 payload, batch size of 400. [3]. **: Counting bitcoin full nodes [29]

Furthermore, the transition from classic BFT SMR to the domain of permissioned blockchains highlight the scalability challenges that this class of protocols faces as well. This is illustrated in Table 2.1, where permissionless platforms like Bitcoin and Algorand tend to scale to a large number of nodes but provide poor throughput and consistency guarantees. Meanwhile, platforms that are based on classic BFT approaches (as is the case with HotStuff) achieve strong consistency, good throughput but low scalability. In practical terms, tackling permissioned blockchains would allow one to verify if Byzantine Leaderless SMR protocols can help with the scalability issues of these applications. Moreover, positive results in the permissioned model can present an opportunity to port this new class of protocols to the permissionless model.

Chapter 3

Leaderless State-Machine Replication: Specification, Properties, Limits

3.1 Introduction

In this chapter we study in-depth the new class of leaderless state-machine replication (Leaderless SMR) protocols and state some of their limits. We provide a theoretical framework that allows one to understand the phenomena described in §2.4 as well as express different Leaderless SMR protocols with it. We define Leaderless SMR and deconstruct it into basic building blocks (§3.2). Further, we introduce a set of desirable properties for Leaderless SMR: (R)eliability, (O)ptimal (L)atency and (L)oad Balancing. Protocols that match all of the ROLL properties are subject to a trade-off between performance and reliability. More precisely, in a system of n processes, the ROLL theorem (§3.3) states that Leaderless SMR protocols are subject to the inequality $2F + f - 1 \leq n$, where $n - F$ is the size of the fast path quorum and f is the maximal number of tolerated failures. A protocol is ROLL-optimal when F and f cannot be improved according to this inequality. We establish that ROLL-optimal protocols are subject to a chaining effect that affect their performance (§3.4). As EPaxos is ROLL-optimal and Mencius not, the chaining effect explains the performance results observed in Figure 2.4. Finally, we discuss the implications of this result (§3.5) then put our work in perspective (§3.6).

3.2 Leaderless SMR

Some recent protocols [104, 106] further push the idea of partially ordered log, as proposed in Generic SMR. In a leaderless state-machine replication (Leaderless SMR) protocol, there is no primary process to arbitrate upon the ordering of commands. Instead, any process may decide a command submitted to the replicated service. A command is stable, and thus executable, once the transitive closure of its predecessors is known locally. As this transitive closure can be cyclic, the log is replaced with a directed graph.

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

3.2.1 Definition

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

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

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

Figure 3.1 depicts an example run of Leaderless SMR that illustrates the above definitions. In this run, process p_1 submits command a, while p_2 submits in order

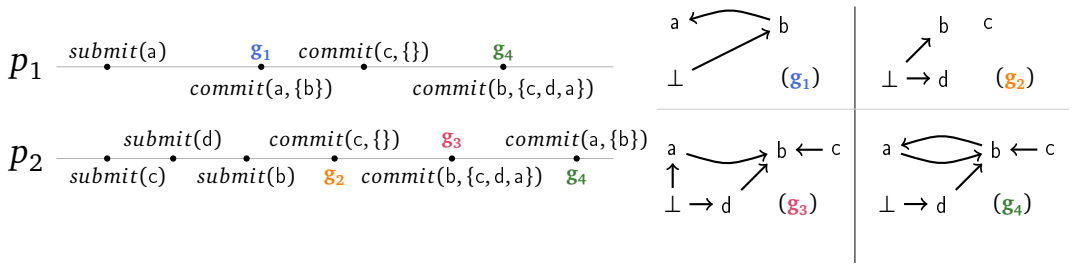


Figure 3.1. An example run of Leaderless SMR – (left) processes p_1 and p_2 submit respectively the commands {a} and {b, c, d}; (right) the dependencies graphs formed at the two processes.

c, d then b. The timeline in Figure 3.1 indicates the timing of these submissions. It also includes events during which process p_1 and p_2 commits commands. For some of these events, we depict the state of the dependency graph at the process (on the right of Figure 3.1). As an example, the two processes obtain the graph g_4 at the end of the run. In this graph, a, b and c are all committed, while d is still pending. We have $deps(a) = \{b\}$ and $deps(b) = \{a, d, c\}$, with both $deps^*(a)$ and $deps^*(b)$ equal to $\{a, b, c, d\}$. Only command c is stable in g_4.

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

Stability: For each command c, there exists D such that if c is stable then $deps(c) = D$.

Consistency: If a and b are both committed and conflicting, then $a \in deps(b)$ or $b \in deps(a)$.

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

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

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

Algorithm 1 Executing command c – code at process p

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

$\mathsf{d} \notin \mathit{deps}^*(\mathsf{c})$; or *(ii)* $\mathsf{c} \in \mathit{deps}^*(\mathsf{d})$, $\mathsf{d} \in \mathit{deps}^*(\mathsf{c})$ and $\mathsf{c} < \mathsf{d}$. Relation $\rightarrow$ defines the *execution order* at a process. If there is a one-way dependency between two commands, Leaderless SMR plays them in the order of their transitive dependencies; otherwise the algorithm breaks the tie using the arbitrary order $<$. This guarantees the following invariant.

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

Generic and Leaderless SMR are strongly similar. In fact, one may show that Generic SMR reduces to Leaderless SMR without requiring any message exchange. This result is stated in Theorem 1 below, and a proof appears in Appendix A.3. Let us observe that such a reduction does not hold between Classic and Generic SMR. Indeed, computing a total order on commuting commands would require processes to communicate.

$\triangleright$ Theorem 1. Generic SMR reduces to Leaderless SMR.

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

3.2.2 Deciding commands

In Leaderless SMR, processes have to agree on the dependencies of stable commands. Thus, a subsequent refinement leads to consider a family of consensus objects $(\mathsf{CONS}_\mathsf{c})_{\mathsf{c} \in \mathscr{C}}$ for that purpose. For some command c, processes use CONS_c to decide either the dependencies of c, or the special value $(\top)$ signaling

that the command is aborted. This agreement is driven by the command *coordinator* (*coord*(c)), a process initially in charge of submitting the command to the replicated state machine. In a run during which there is no failure and the failure detector behaves perfectly, that is a *nice run*, only *coord*(c) calls CONS_c.

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

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

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

Visibility: If $\mathsf{c} \asymp \mathsf{c}'$ and $D' \in 2^{\mathscr{C}}$, then $\mathsf{c} \in D'$ or $\mathsf{c}' \in D$.

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

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

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

Algorithm 2 Deciding a command c – code at process p

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

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

Below, we establish the correctness of Algorithm 2. A proof appears in Appendix A.3.

▷ Theorem 2. Algorithm 2 implements Leaderless SMR.

3.2.3 Examples

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

Rotating coordinator. For starters, let us consider a rotating coordinator algorithm (e.g., [133]). In this class of protocols, commands are ordered *a priori* by some relation $\ll$. Such an ordering is usually defined by timestamping commands at each coordinator and breaking ties with the process identities. When $\mathit{coord}(\mathsf{c})$ calls $\mathsf{DDS}.\mathit{announce}(c)$, the service returns a pair (D, false), where D are all the commands prior to c according to $\ll$. Upon recovering a command, the DDS service simply suggests to abort it.

Clock-RSM. This protocol [57] improves on the above schema by introducing a fast path. It also uses physical clocks to speed-up the stabilization of committed commands. Once a command is associated to a timestamp, its coordinator broadcasts this information to the other processes in the system. When it receives such a message, a process waits until its local clock passes the command's timestamp to reply. Once a majority of processes have replied, the DDS service informs the coordinator that the fast path was taken.

Mencius. The above two protocols require a committed command to wait all its predecessors according to $\ll$. Clock-RSM propagates in the background the physical clock of each process. A command gets stable once the clocks of all the processes is higher than its timestamp. Differently, Mencius [104] aborts prior pending commands at the time the command is submitted. In detail, *announce*(c) first approximates D as all the commands prior to c according to $\ll$. Then, command c is broadcast to all the processes in the system. Upon receiving such a message, a process q computes all the commands d smaller than c it is coordinating. If d is not already announced, q stores that d will be aborted. Then, q sends d back to *coord*(c) that removes it from D. The DDS service returns (D, f) with f set to *true* if *coord*(c) received a message from everybody. Upon recovering c, if the command was received the over-approximation based on $\ll$ is returned together with the flag *false*. In case c is unknown, the DDS service suggests to abort it.

EPaxos. In [106], the authors present Egalitarian Paxos (EPaxos), a family of efficient Leaderless SMR protocols. For simplicity, we next consider the variation which does not involve sequence numbers. To announce a command c, the coordinator broadcasts it to a quorum of processes. Each process p computes (and records) the set of commands D_p conflicting with c it has seen so far. A call to *announce*(c) returns $(\cup_p D_p, b)$, with b set to *true* iff processes spontaneously agree on dependencies (i.e., for any p, q, $D_p = D_q$).

When a process in the initial quorum is slow or a recovery occurs, c is broadcast to everybody. The caller then awaits for a majority quorum to answer and returns $(D, false)$ such that if at least $\lceil \frac{n+1}{2} \rceil$ processes answer the same set of conflicts for c, then D is set to this value (with $n = 2f + 1$). Alternatively, if at least one process knows c, the union of the response values is taken. Otherwise, the DDS service suggests to abort c.

Caesar. To avoid cycles in the dependency graph, Caesar [16] orders commands using logical timestamps. Upon submitting a command c, the coordinator timestamps it with its logical clock then it executes a broadcast. As with EPaxos, when it receives c a process p computes the conflicting commands D_p received so far. Then, it awaits until there is no conflicting command d with a higher timestamp than c such that c $\notin$ *deps*(d). If such a command exists, p replies to the coordinator that the fast path cannot be taken. The DDS service returns $(\cup_p D_p, b)$, where $b = true$ iff no process disables the fast path.

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

3.2.4 Core properties

State machine replication helps to mask failures and asynchrony in a distributed system. As a consequence, a first property of interest is the largest number of failures (parameter f) tolerated by a protocol. After f failures, the protocol may not guarantee any progress.[1]

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

Leaderless SMR protocols exploit the absence of contention on the replicated service to boost performance. In particular, some protocols are able to execute a command after a single round-trip, which is clearly optimal [92]. To ensure this property, the fast path is taken when there is no concurrent conflicting command. Moreover, the command stabilizes right away, requiring that the DDS service returns only submitted commands.

(*Optimal Latency*) During a nice run, every call to *announce*(c) returns a tuple (D, b) after two message delays such that *(i)* if there is no concurrent conflicting command to c, then b is set to *true*, *(ii)* $D \in 2^{\mathscr{C}}$, and *(iii)* for every $d \in D$, d was announced before.

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

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

The previous properties are formally defined in Appendix B.1. Table 3.1 indicates how they are implemented by well-known leaderless protocols. The columns 'Reliability' and 'Load Balancing' detail respectively the maximum number of failures tolerated by the protocol and the size of the fast path quorum.

[1] When f failures occur, the system configuration must change to tolerate subsequent ones. If data is persisted (as in Paxos [88]), the protocol simply stops when more than f failures occurs and awaits that faulty processes are back online.

Protocols	***Properties***			
	Load Balancing ($n-F$)	Reliability (f)	Optimal Latency	ROLL-optimal
Rotating coord.	0	Min	×	×
Clock-RSM [57]	n	Min	×	×
Mencius [104]	n	Min	√	×
Caesar [16]	$\lceil \frac{3n}{4} \rceil$	Min	√	×
EPaxos [106]	LMaj	Min	√	if $n = 2f+1$
Alvin [131]	LMaj	Min	√	if $n = 2f+1$
Atlas [62]	$\lfloor \frac{n}{2} \rfloor + f$	any	√	if $n \in 2\mathbb{N} \cup \{3\} \wedge f = 1$

Table 3.1. The properties of several leaderless SMR protocols – Min stands for a minority of replicas ($\lfloor \frac{n-1}{2} \rfloor$), Maj a majority ($\lceil \frac{n+1}{2} \rceil$), and LMaj a large majority ($\lfloor \frac{3n}{4} \rfloor$).

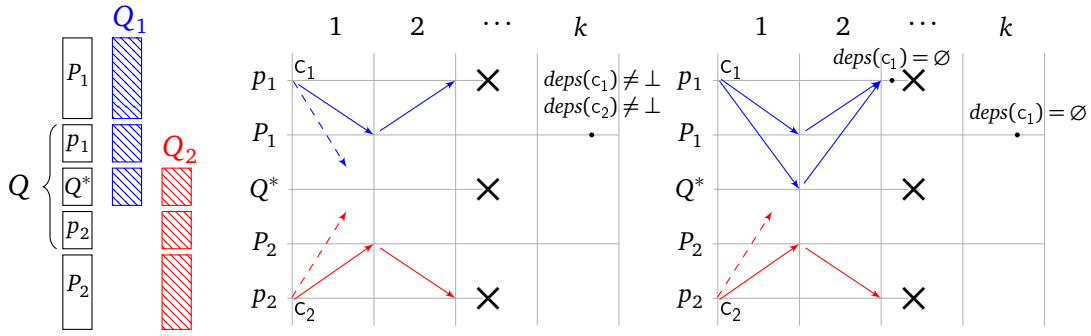
Notice that by CAP [71], we have $F, f \leq \lfloor \frac{n-1}{2} \rfloor$ when the protocol matches all of the properties. Table 3.1 also mentions the optimality of each protocol with respect to the ROLL theorem. This theorem is stated in the next section and establishes a trade-off between fault-tolerance and performance in Leaderless SMR.

3.3 The ROLL theorem

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

Unfortunately, we show that there is no free-lunch and that an optimization choice must be made. The ROLL theorem below establishes that $2F + f - 1 \leq n$ must hold. This inequality captures that every protocol must trade scalability for fault-tolerance. EPaxos [106] and Atlas [62] illustrate the two ends of the spectrum of solutions (see Table 3.1). EPaxos supports that any minority of processes may fail, but requires large quorums. Atlas typically uses small fast path quorums ($\lfloor \frac{n}{2} \rfloor + f$), but exactly handles at most f failures.

Below, we state the ROLL theorem and provide a sketch of proof illustrated



(a) Quorums in use.

(b) Run λ_3.

(c) Run λ_4.

Figure 3.2. Illustration of Theorem 3 – slow messages are omitted.

in Figure 3.2. A formal treatment appears in Appendix B.

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

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

As depicted in Figure 3.2a, the distributed system is partitioned into three sets: P_1 and P_2 are two disjoints sets of $F-1$ processes, and the remaining $n-2(F-1)$ processes form Q. The CAP impossibility result [71] tells us that $2F < n$. As a consequence, there exist at least two distinct processes p_1 and p_2 in Q. We define Q_1 and Q_2 as respectively $P_1 \cup Q \setminus \{p_2\}$ and $P_2 \cup Q \setminus \{p_1\}$. The set Q^* equals $Q \setminus \{p_1, p_2\}$.

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

Then, consider a run λ_3 in which p_1 and p_2 submit concurrently commands c_1 and c_2. This run is illustrated in Figure 3.2b. At the end of the first round, the processes in P_1 (respectively, P_2) receive the same messages as in λ_1 (resp., λ_2). At the start of the second round, they reply to respectively p_1 and p_2 as in λ_1 and λ_2. All the other messages sent in the first two rounds are arbitrarily slow. The processes in Q crash at the end of the second round. By Reliability and as $f \geq |Q|$, the commands c_1 and c_2 are stable in λ_3. Let k be the first round at

which the two commands are stable at some process $p \in P_1 \cup P_2$.

We now build an admissible run λ_4 of $\mathscr{P}$ as follows. The failure pattern and failure detector history are the same as in λ_3. Commands c_1 and c_2 are submitted concurrently at the start of λ_4, as in λ_3. In the first two rounds, P_1 receives the same messages as in λ_1 while P_2 receives the same messages as in λ_3. The other messages exchanged during the first two rounds are arbitrarily slow. Figure 3.2c depicts run λ_4.

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

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

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

3.3.1 Optimality

A protocol is *ROLL-optimal* when the parameters F and f cannot be improved according to Theorem 3. In other words, they belong to the skyline of solutions [31]. As an example, when the system consists of 5 processes, there is a single such tuple $(F, f) = (2, 2)$. With $n = 7$, there are two tuples in the skyline, $(2, 3)$ and $(3, 2)$. The first one is attained by EPaxos, while Atlas offers the almost optimal solution $(3, 1)$ (see Table 3.1).

For each protocol, Table 3.1 lists the conditions under which ROLL-optimality is attained. EPaxos and Alvin are both optimal under the assumption that $n = 2f + 1$. Atlas adjusts the fast path quorums to the value of f, requiring $\lfloor \frac{n}{2} \rfloor + f$ processes to participate. This is optimal when $f = 1$ and either n is even or equals to 3. In the general case, the protocol is within $O(f)$ of the optimal value. As it uses classical Fast Paxos quorums, Caesar is not ROLL-optimal. This is also the case of protocols that contact all of the replicas to make progress, such as

Mencius and Clock-RSM. To the best of our knowledge, no protocol is optimal in the general case.

In the next section, we show that ROLL-optimality has a price. More precisely, we establish that by being optimal, a protocol may create an arbitrarily long chain of commands, even during a nice run. This chaining effect may affect adversely the performance of the protocol. We discuss measures of mitigation in §3.5.

3.4 Chaining effect

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

At first glance, one could think that this situation arises from the asynchrony of the distributed system. As illustrated in Figure 2.4, this is not the case. We establish that such an effect may occur during "almost" synchronous runs.

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

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

3.4.1 Notion of chain

A chain is a sequence of commands $\mathsf{c}_1 \dots \mathsf{c}_n$ such that for any two consecutive commands $(\mathsf{c}_i, \mathsf{c}_{i+1})$ in the chain, $\mathsf{c}_i \in \mathit{deps}(\mathsf{c}_{i+1})$ at some process. Two consecutive commands (c, d) in a chain form a *link*. For instance, in the dependency graph $\mathbf{g_4}$ (see Figure 3.1), cba is a chain.

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

3.4.2 A measure of asynchrony

In a synchronous system [100], processes execute rounds in lock-step. During a round, the messages sent at the beginning are received at the end (provided there is no failure). On the other hand, a partially synchronous system may delay messages for an arbitrary amount of time. In this model, we propose to measure asynchrony by looking at the overlaps between the exchanges of messages. The larger the overlap is, the more asynchronous is the run.

To illustrate this idea, consider the run depicted in Figure 3.3. During this run, a red message is sent from p_5 to p_4 (bottom left corner of the figure). In the same amount of time p_1 sends a blue message to p_2 which is followed by a green message to p_4. To characterize such an asynchrony, we shall say that the run is 2-asynchronous. This notion is precisely defined below.

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

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

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

3.4.3 Result statement

The theorem below establishes that a ROLL-optimal protocol may create a live chain of arbitrary size during a 2-asynchronous nice run. The full proof appears in Appendix B.4.

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

Proof. (Sketch) The theorem is proved by adding inductively a new link to a live chain of commands created during a nice run. It is illustrated in Figure 3.3 for a system of five processes when $k = 7$.

The proof is based on the following two key observations about ROLL-optimal protocols. First, during a nice run, the coordinator of a command never rotates.

[2] That is, neither $recv(m)$ precedes $send(m')$, nor $recv(m')$ precedes $send(m)$ in real-time.

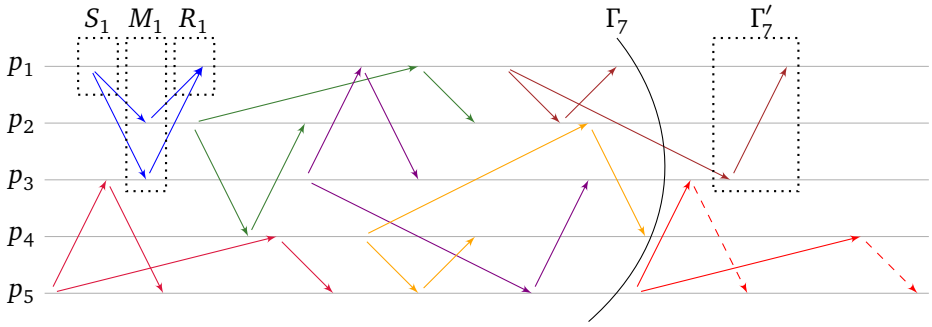


Figure 3.3. Theorem 4 for $n = 5$ and $k = 7$. The chain $c_7c_6c_5c_4c_3c_2c_1$ is formed in σ_7. Illustrating the steps S_1, M_1 and R_1 for command c_1, the prefix Γ_7 of σ_7, and the steps Γ'_7.

As a consequence, the return value of the DDS service at the coordinator is always the stable value of $deps(c)$. Second, as the protocol satisfies the ROLL properties, a call to $announce(c)$ consists of sending a set of requests to the fast path quorum and receiving a set of replies. As a consequence, its execution can be split into the steps $S_cM_cR_c$, where (S_c) are the steps taken from announcing c to the sending of the last request at the coordinator; (R_c) are the steps taken by $coord(c)$ after receiving the first reply until the announcement returns; and (M_c) are the steps taken during the announcement of c which are neither in S_c, nor in R_c. By Optimal Latency, this sequence of steps do not create pending messages. As an illustration, the steps S_1, M_1 and R_1 taken to announce command c_1 are depicted in Figure 3.3.

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

Given a sequence of steps λ and a set of processes Q, let us note $\lambda|Q$ the subsequence of steps by Q in λ. We establish that at rank $i > 0$ the following property $\mathfrak{P}(i)$ holds: There exists a 2-asynchronous run σ_i of the form $\Gamma_i S_i(M_i|P_i)\Gamma'_i(M_i|Q_i \setminus P_i)R_i$ such that *(1)* $proc(\Gamma'_i) \cap Q_i = P_i$; *(2)* every path in Γ'_i is as most of size one; *(3)* no message is pending in σ_i; and *(4)* σ_i contains a chain $c_ic_{i-1}\cdots c_1$. Figure 3.3 depicts the run σ_7, its prefix Γ_7 and the steps Γ'_7.

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

At rank $i+1$, item (1) is proved with appropriate definitions of the quorums (Q_i and Q_{i+1}), and the sub-quorum (P_i). For instance, in Figure 3.3, the command c_1 and c_2 have respectively $\{p_1, p_2, p_3\}$ and $\{p_3, p_4, p_5\}$ for fast path

quorums. The sub-quorum P_2 is set to the intersection of Q_1 and Q_2, that is $\{p_3\}$. Item (2) follows from the definition of Γ'_{i+1}. The Load-Balancing property implies that (3) holds. A case analysis can then show that σ_{i+1} is 2-asynchronous. It relies on the fact that the $(SMR)_{i+1}$ steps create no pending message and the induction property $\mathfrak{P}(i)$.

To prove that a new link was added, we show that σ_{i+1} is indistinguishable to $coord(c_i)$ to a run in which c_{i+1} gets committed while missing c_i. Going back to Figure 3.3, observe that the coordinator of c_6 does not know that the replies of p_2 for command c_5 causally precedes the replies of p_4 to c_7. As a consequence, it must add c_7 to the return value of $DDS.announce(c_6)$.

Finally, to obtain a live chain of size k, it suffices to consider the prefix of σ_{i+1} which does not contain the replies of the fast path quorum. In Figure 3.3, this corresponds to omitting the dashed messages that contain the reply to the announcement of c_7 □

3.5 Discussion

Leaderless SMR offers appealing properties with respect to leader-driven approaches. Protocols are faster in the best case, suffer from no downtime when the leader fails, and distribute the load among participants. For instance, Figure 2.4 shows that EPaxos is strictly better than Paxos when there is no conflict. However, the latency of a command is strongly related to its dependencies in this family of SMR protocols. Going back to Figure 2.4, the bivariate correlation between the latency of a command and the size of the batch with which it executes is greater than 0.7.

Several approaches are possible to mitigate the chaining effect established in Theorem 4. Moraru et al. [106] propose that pure writes (i.e., commands having no response value) return once they are committed.[3] In [62], the authors observe that as each read executes at a single process, they can be excluded from the computation of dependencies. A third possibility is to wait until prior commands return before announcing a new one. However, in this case, it is possible to extend Theorem 4 by rotating the command coordinators to establish that a chain of size n can form.

In ROLL, the Load-Balancing and Optimal-Latency properties constrain the form of the DDS service. More precisely, in a contention-free case, executing the

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

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

3.6 Related work

Protocols. Early leaderless solutions to SMR include rotating coordinators and deterministic merge, aka. collision-fast, protocols. We cover the first class of protocols in §3.2.3. In a collision-fast protocol [9, 121], processes replicate an infinite array of vector consensus instances. Each vector consensus corresponds to a round. During a round, each process proposes a command (or a batch) to its consensus instance in the vector. If the process is in late, its peers may take over the instance and propose an empty batch of commands. Commands are executed according to their round numbers, applying an arbitrary ordering per round. The size of the vector can change dynamically, adapting to network conditions and/or the application workload. This technique is also used in Paxos Commit [77].

When the ordering is fixed beforehand, processes must advance at the same pace. To fix this issue, Mencius [104] includes a piggy-back mechanism that allows a process to bail out its instances (i.e., proposing implicitly an empty batch). Clock-RSM [57] follows a similar schema, using physical clocks to bypass explicit synchronization in the good cases.

With the above protocols, commands still get delayed by slow processes. Avoiding this so-called delayed commit problem [104] requires to dynamically discover dependencies at runtime. This is the approach introduced in Zieliński's optimistic generic broadcast [141] and EPaxos [106]. Here, as well as in [62], replicas agree on a fully-fledged dependency graph. Caesar [16] uses timestamps to avoid cycles in the graph. However, even in contention-free cases, committing a command can take two round trips. In our classification (see Table 3.1), this protocol does not have Optimal Latency.

Deconstruction. In [30], the authors introduce the dependency-set and map-

agreement algorithms. The two services allow respectively to gather dependencies and agree upon them. A similar decomposition is proposed in [137]. Compared to these prior works, our framework includes the notion of fast path and distinguishes committed and stable commands. An agreement between the processes is necessary only eventually and on the stable part of the dependency graph. This difference allows to capture a wider spectrum of protocols. Our dependency discovery service (DDS) is reminiscent of an adopt-commit object [65] that allows processes to reach a weak agreement. In our case, when the fast path flag is set, processes may disagree on at most the aborted dependencies of a command.

Complexity. Multiple works study the complexity of consensus, the key underlying building block of SMR. Lamport [92] proves several lower bounds on the time and space complexity of this abstraction. The Hyperfast Learning theorem establishes that consensus requires one round-trip in the general case. This explains why we call optimal protocols that return after two message delays. The Fast Learning theorem requires that $n > 2F + f$. This result explains the trade-off between fault-tolerance and performance in Fast Paxos [91]. However, it does not readily apply to Leaderless SMR because only coordinator-centric quorums are fast in that case. For instance, EPaxos is able to run with $F = 1$ and $f = 1$ in a 3-process system. The ROLL theorem (§3.3) accurately captures this difference.

Traditional complexity measures for SMR and consensus (e.g., the latency degree [120]) consider contention-free and/or perfectly synchronous scenarios. In [15], the authors study the complexity of SMR over long runs. The paper shows that completing an SMR command can be more expensive than solving a consensus instance. Their complexity measure is different from ours and given in terms of synchronous rounds. In §3.4, we show that in an almost synchronous scenario, contention may create arbitrarily long chains in Leaderless SMR. We discuss mitigation measures in §3.5.

Chapter 4

Byzantine Leaderless SMR

4.1 Introduction

In this chapter we present the first framework for Leaderless Byzantine State-Machine Replication and we propose instantiations for it. The first instantiation of interest is a Byzantine version of EPaxos [106]. The second instantiation is a new protocol that we call *Wintermute* (§4.3.3). This protocol has an overall better asymptotic behavior than traditional BFT SMR protocols. More specifically, it has a lower load and message complexity than prior art, allowing it to side-step the scalability problems inherent to BFT solutions as found, e.g., in the context of permissioned blockchains.

4.1.1 Key Observations

A new observation can be made once one looks to SMR through the lenses of the leaderless approach presented in Chapter 3. By breaking down ordering into the DDS and CONS services, a new level of scalability can be reached, which is unheard of in terms of state-machine replication. More specifically, as the two services scale separately, it is possible to populate the DDS service with a large amount of processes, while maintaining an order of magnitude less to run CONS (ideally the bare minimum, that is $3f + 1$).

To motivate the above idea, consider for instance PBFT (§2.2.5). In this protocol, the leader continuously contacts all the replicas, waiting for a quorum of replies. In contrast, with our approach, the coordinator of a command in the DDS service has a one round-trip of communication and *exclusively* to a single quorum of replicas. Moreover, the Leaderless SMR decomposition allows different implementations of the DDS and CONS services, yielding to new protocols

that that can be tailored for different use cases.

In combination with the above observation, another key observation can be made once we analyse the adversary in the permissioned model for permissioned blockchains. In the vein of the XFT model (shorthand for *cross fault tolerance*) introduced in [97], where through real world experiments the authors argue for the weakening of the power of Byzantine processes, we observe similarly for permissioned blockchains that is reasonable to assume a small number of Byzantine failures, as the adversary can be weakened too. Such assumption can be sustained by the fact that under the permissioned model a set of arguments and techniques can be applied to curb the power of Byzantine processes.

Notably, as the processes are not anonymous and in fact, generally controlled by companies that have stake to lose, accountability [79, 43, 114] can be a powerful tool. Further, network churn is moderate as the procedure to join and leave the network is costly. Moreover, doing a coordinated attack to many companies is quite hard due to their different architectures and safety mechanisms. And last, very much in the style of permissionless blockchains, Byzantine behavior can be curbed by the addition of a cryptocurrency on top of the blockchain to diminish Byzantine behavior through economic incentives.

The combination of the two previous ideas make the leaderless approach advantageous for use in permissioned blockchains[1]. It allows one to balance, at the same time: scalability, trust and performance.

4.1.2 Primer on the results

We present now a primer on the results by comparing our Leaderless BFT SMR protocol Wintermute with other BFT SMR protocols considering a specific set of metrics explained next.

We cast all protocols into the proxy model used in Chapter 3 (i.e. clients are not modeled) and consider the best-case scenario, that is, the critical path to a decision. Informally, the critical path to a decision of a command c consists of a nice run λ that starts from the initial state until command c is decided. Using the operators presented in Appendix A, we define formally that a critical path for a command c is: $(\lambda| \geq submit(c))| \leq decide_{coord(p)}(c)$.

Latency (Δ) represents message delays and *Message Complexity* (μ) represents message complexity. Both metrics have been previously defined in Appendix B. *Authenticator complexity* (α) is a metric borrowed from [139] that after

[1]Its important to remember that the commutativity rate between transactions might be high in blockchains which benefits the leaderless approach even further

adaption to our means is defined as: an *authenticator* is a signature or a partial signature and the complexity is the sum, over all processes $p \in \Pi$, of the number of authenticators received by a process p in the critical path.

Differently from the definition of load of a Byzantine quorum system we define load (l) as follows: the worst probability of a process being part of the ordering of a command.

For PBFT, we consider the three-phase protocol with: PRE-PREPARE, PREPARE, COMMIT and REPLY. For HotStuff, we consider the phases: PREPARE, PRE-COMMIT, COMMIT, DECIDE, with two message delays in each phase besides DECIDE with one message delay. For SBFT, assuming fast path, we count the message delays in the phases: PRE-PREPARE, SIGN-SHARE, FULL-COMMIT-PROOF, SIGN-STATE, EXECUTE-ACK. For Pompe, we count four message delays during the ordering phase and two more for the consensus phase (processes still need to run consensus though, making the final cost to reach a decision even higher than six). We explain in §4.4 how we reach the metrics with Wintermute.

	Metrics			
Protocols	Δ	μ	α	l
PBFT [41]	4	$O(n^2)$	$O(n^2)$	1
HotStuff [139]	7	$O(n)$	$O(n)$	1
SBFT [74]	5	$O(n)$	$O(n)$	1
Pompe [140]	6+*	$O(n)$	$O(n)$	1
Wintermute	6	$O(\sqrt{fn})$	$O(\sqrt{fn})$	$O(\sqrt{f/n})$

Table 4.1. Performance of a collection of BFT protocols. In the critical path, always during nice runs, from a quiescent state (i.e. best case). Δ: Latency (message delays), α: Authenticators complexity, μ: Message Complexity, l: Load. * The actual message delay for a decision will be higher depending on the byzantine consensus protocol used.

4.2 Byzantizing Leaderless SMR

In this section we show how to make Leaderless Byzantine SMR a reality and at the same time answer the question posed in [140] about how EPaxos' properties can be guaranteed in a Byzantine faul-tolerant model, by casting a variation of the protocol into our framework.

We begin with the definition of Byzantine Leaderless SMR, or BLSMR for short. The abstraction is identical as the one described in §3.2, that is a dis-

tributed automata with operations *submit* and *commit*. It is defined as the conjunction of the properties below.

Validity: At a correct process, a command is decided once and only if it was submitted.

Stability: For each command c, there exists D such that at a correct process, if c is stable then $deps(c) = D$.

Consistency: At a correct process, if a and b are both conflicting and committed, then $a \in deps(b)$ or $b \in deps(a)$.

One may understand these properties as a non-uniform variation of the ones given in §3.2.1. This is similar to the re-writing of the SMR properties in the Byzantine model by Castro and Liskov [41] (see §2.2).

Linearizable objects

Constructing a linearizable shared object atop BLSMR is achievable in a similar manner as atop LSMR. We follow the approach presented in [63, Algorithm 5], with a few changes to accommodate Byzantine failures. The algorithm is sketched below. The interested reader may consult Appendix C for the full details.

Consider that commands are submitted by a set of client processes—that is for each command c, $client(c)$ denotes the client of c. Replicas execute commands once they are stable, accordingly to Algorithm 1. Once a command is executed, its response value is returned to $client(c)$. Due to the presence of Byzantine replicas, a client waits $f + 1$ such responses before taking it into account.

This construction works because *(i)* the properties of BLSMR ensures that every response value at a correct process is linearizable, and *(ii)* because $f + 1$ responses include the one of a correct process, a client only consider such values.

At first, the requirements to implement BLSMR look simple. One might think that in order to achieve a Byzantine-safe version of Algorithm 2 one needs only Byzantine versions of both CONS and DDS. In fact, a Byzantine-hardened connection between the two services is also required for safety. Indeed, one may observe in line 3 of Algorithm 2 that we cannot trust the process in charge of running the DDS service. This process can misbehave and break consistency by forwarding a malformed D to the processes running the CONS service. For instance, consider two conflicting commands c, d and the calls $announce_p(c)$,

$announce_q(\mathsf{d})$. If process p is Byzantine it may omit d from the returning of its call, breaking consistency. Moreover, creating a Byzantine version of the DDS service is tricky. As will be explored later, the representation of dependencies plays an essential role in the process.

4.2.1 Notion of Trusted Service

Trust management is at the core of Byzantine SMR. Indeed, one may observe that both in PBFT [41] and HotStuff [139], the algorithm manipulates *certificates*, or *quorum certificates*, authenticated collections of messages which allow processes to verify the assertions made by the leader and other processes.

In the seminal work of Cachin et al. in [37], the authors formalize how such authenticated collections of messages can be used to create a protocol that is *Verifiable*. They present for example a *Verifiable Consistent Broadcast* (VCBC) protocol, a form of *reliable broadcast* [32] protocol modified such that it doesn't require that two processes deliver the payload message, but require that the consistency property be kept among the actually delivered messages. The protocol is made verifiable through the use of threshold cryptography, just like in HotStuff. A sender broadcasts a message m to all processes and hopes for a threshold of them to partially sign it. Once it receives the partial signatures, the sender combines them in a threshold signature on m and sends such signed message to all processes. A protocol process that was not yet in a state to deliver m, can now be convinced by the threshold signature on m and can safely deliver the message.

In [37] as well, the authors propose the property of *external validity* for the multi-valued Byzantine Agreement problem. In this multi-valued validated asynchronous Byzantine Agreement, each process takes a value as input and decides one of the values as output, as long as the decided output satisfies a global *predicate* that is determined by the particular application and known to all processes.

The external validity property is important because it allows the connection between protocols for VCBC and Byzantine Agreement. Such ideas are present in the abstraction *Validated and Provable Consensus* (VPC) found in [125] and based on [37]. In VPC, *Validated*, means that the protocol receives a predicate γ together with a proposed value - which any decided value must satisfy. By *Provable* it means that the protocol generates a cryptographic proof Γ that certifies that a value v was decided in a certain consensus instance.

Fundamentally, these constructs presented bound the effects of Byzantine nodes on the distributed system and allow the connection between different services. We need that CONS be an abstraction like VPC and require it to be connected to DDS. The previous works point the direction one must take but

do not present a generic way to transform a normal service into a verifiable and provable service.

Thus, in this section we propose to abstract the management of trust through the notion of trusted service. Our idea is to have a generic-enough specification of this class of service such that it can be instantiated with different cryptosystems, or even no cryptography at all (e.g. information theory based approach). Our definition just requires the ability to generate a certificate, or *proof*, of an action taken by a process or a set of processes.

Preliminaries

To include trust in our algorithms, we need to embed it in the values they manipulate. An algorithm may use this trust to verify that the value is not crafted by some malicious node. For starters, we shall consider that this verification is stateless.

We say that a value x is *provable* if x embeds a proof, that is $x.\Gamma \in \mathscr{E}^*$, where $\mathscr{E}^*$ is the universe of proofs. For some pair of provable values (x, y), a relation f is *verifiable* if there exists a boolean function $check_f$, called *the verifier of* f, such that $check_f(x, y)$ implies that $f(x) = y$. Next, given a verifiable relation f and two provable values x and y , we define a trusted assignment operator. This operator provides syntaxic glue to check whether the values are mapped with relation f, before making the assignment. If this is not the case, an error is raised. Formally,

$$var \overset{f,x}{\longleftarrow} y \triangleq \begin{array}{l} \textbf{if } check_f(x, y) \\ \textbf{then } var \leftarrow y \\ \textbf{else error} \end{array}$$

To illustrate the above notions, consider the notion of verifiable random function, as defined in §2.2.3. We map the evaluator $\mathrm{VRF}.Eval_{SK}(x)$ and the verifier $\mathrm{VRF}.V_{VK}(\pi, x, y)$ to respectively the notion of verifiable function and verifier defined above. For the verifier, the proof π is attached to both x and y to make them provable values, that is $x.\Gamma = y.\Gamma = \pi$.

A verifiable relation might be considered as a stateless automata, that is it does not keep a state from one invocation to another. Thus, to define a trusted service, we need to make several extensions. First, a service can be stateful, which means that the checking the existence of a mapping is based on an history of the service, and not just its inputs and outputs. Second, a service is a protocol, that is a distributed automata and not a locally computable function. Nonetheless, its verifier is still a locally computable function. Last, as we need

to compose services one with another, we need to introduce such a notion. All these extensions are detailed hereafter.

In the context of this work, we consider a service to be a linearizable non-deterministic shared object. For instance, consensus is modeled as an object offering the operation *propose* at its interface. When a service S exposes a single operation, we use the shorthand $S(x) = y$ to indicate that S returns the output y from input x. (The run to which we refer is omitted when appropriate.)

Definitions

Consider a provable value y and a service S. Service S is *verifiable* when there exists a boolean function $check_S(x, y)$ with x and y two provable values, called the verifier of S, such that if $check_S(x, y)$ holds then $S(x) = y$ is true. Let x be a provable value and H be a verifiable service. A service S is *verifying wrt.* $(H, \hat{x})$ when $S(x) = y$ implies that $check_H(\hat{x}, x)$ holds. A service S is *trusted* wrt. some verifiable service H and provable input x when S is verifiable and verifying wrt. (H, x). We extend naturally the trusted assignment operator to work with services. In detail, for S verifiable,

$$var \overset{S,x}{\longleftarrow} y \triangleq \begin{array}{l} \textbf{if } check_S(x, y) \\ \textbf{then } var \leftarrow y \\ \textbf{else error} \end{array}$$

To illustrate the above definitions, we can consider how to modify PBFT to make it a trusted service. The service is trusted if we bound the protocol to only work on correctly signed commands (its verifying in relation to the PKI service) and verifiable because we can provide the verifying function $check_{PBFT}(x, y)$, where x contains a signed state-machine command c and a view i and y contains the set of $2f + 1$ signed COMMIT messages that triggered the decision of command c in view i.

4.2.2 Algorithm

Building upon the previous definitions, Algorithm 3 depicts a construction to decide commands in a leaderless manner when processes are Byzantine. This algorithm uses trusted variations of the dependency discovery (DDS) and consensus (CONS_c) services. In detail, Algorithm 3 relies on the following two services:

Trust: DDS is a trusted service wrt. to $(id_{\mathscr{C}}, \_)$, where $id_{\mathscr{C}}$ returns *true* iff its input is a command. For every command c, $\mathsf{CONS_c}$ is a trusted service wrt. $(\mathsf{DDS}, \mathsf{c})$

Any Byzantine consensus service that provides primitives to agree on a value can be used to implement CONS_c, as long as this service ensures that its (provable) input values are verified. In practice this is usually already the case, since protocols like HotStuff sign state-machine commands and produce quorum certificates and famous implementations like BFT-SMaRt [24] are built on top of abstractions like VPC [125] (matching our definitions).

Interestingly, the fact that CONS_c is verifying DDS is only required to ensure progress. This prevents Byzantine nodes to push erroneous values to CONS_c, preventing a decision on c.

In Algorithm 3, the text in blue correspond to the differences with Algorithm 2. Notably, Algorithm 3 relies on the trusted assignment operator which was defined in the previous section. This encapsulates in a precise and well-defined manner the necessity to use a Byzantine versions of DDS and CONS_c, as well as the connection between these two services.

When it receives a message indicating a decision for some command, Algorithm 3 makes use of a verifiable function f to check this decision. This function takes as input a command (c), a set of dependencies (D) and a flag (b), and it returns the dependencies D. Its verifier is defined below:

$$check_f((c, D, b), D) \triangleq \textbf{return if } check_{\mathsf{DDS}}(c, (D, b)) \textbf{ then } b \vee check_{\mathsf{CONS}_c}(D, D) \textbf{ else } false$$

Algorithm 3 Deciding a command c – code at process p

1: $submit(c)$:=
2: **pre:** $p = coord(c) \vee coord(c) \in \mathscr{D}$
3: **eff:** $(D, b) \xleftarrow{DDS,c} \mathsf{DDS}.announce(c)$
4: **if** $b = false$ **then** $D \xleftarrow{\mathsf{CONS}_c,D} \mathsf{CONS}_c.propose(D)$
5: $broadcast(c, D, b)$ *to* Π
6:
7: **when** $recv(c, D, b)$
8: **pre:** $deps(c) = \bot$
9: **eff:** $deps(c) \xleftarrow{f,(c,D,b)} D$

In the propositions that follow, we prove the correctness of Algorithm 3. To this end, let us denote with λ a run of this algorithm. We first characterize the necessary trust to execute successfully the assignment at line 9.

Proposition 1. *If* $check_{id_{\mathscr{C}}}(\mathsf{a}, true)$ *returns true at process p, then* a *is a submitted command in* λ*.*

Proof. As $check_{id_{\mathscr{C}}}(\mathsf{a}, true)$ returns *true*, we have $\mathsf{a} \in \mathscr{C}$. In §2.2.2, we require that p does not have access to $\mathscr{C}$ at the start of λ and that commands are not craftable. Thus, since clients are correct, a was necessarily submitted. □

Proposition 2. *During* λ*, consider that a correct process p executes the trusted assignment at line 9 in Algorithm 3, that is* $deps(\mathsf{c}) \xleftarrow{f,(\mathsf{c},D,b)} D$*. If this assignment does not raise an error then* (i) *command* a *was submitted, and* (ii) DDS *is called with input* a *in* λ *and returns* (D, b)*.*

Proof. If the trusted assignment does not return an error, then $check_f((c, D, b), D)$ returns *true* at p. At the light of the definition of $check_f$, $check_{\mathsf{DDS}}(c, (D, b))$ equals *true*. Service DDS is trusted with respect to $(id_{\mathscr{C}}, \_)$. As a consequence, *(i)* DDS is verifiable. Hence, as $check_{\mathsf{DDS}}(c, (D, b))$ returns *true* at process p, DDS is called with input c in λ and returns a tuple (D, b). *(ii)* DDS is verifying with respect to $(id_{\mathscr{C}}, \_)$. Applying Proposition 1, command a was submitted. □

▷ Theorem 5. Algorithm 3 implements a reliable Byzantine Leaderless SMR.

Proof. We consider in order all the properties of BLSMR. Again, let λ be a run of Algorithm 3.

(Validity) If command a is decided at a correct process p, then $deps(\mathsf{a}) \neq \bot$ holds at p. Necessarily p executes line 9. Due to the precondition at line 8, this happens at most once. Applying Proposition 2, a is submitted.

(Consistency) Let a and b be two conflicting committed commands at some correct process p. By definition, the two commands are decided when $D = deps(\mathsf{a}), D' = deps(\mathsf{b}) \in 2^{\mathscr{C}}$ holds at p. Applying Proposition 2, we know that DDS returns D and D' when calls respectively with command a and b in λ. By the Visibility property, necessarily either $\mathsf{a} \in deps(\mathsf{b})$, or $b \in deps(\mathsf{a})$, as required.

(Stability) Consider a command a and two correct processes p and q. A decision about a takes place at line 9. It uses the pair (D, b) extracted from an incoming message at line 7. Let respectively (E, f) and (E', f') be the value of these variables at p and q when this happens. Observe that $deps(\mathsf{a})$ may only decrease, over time. Name E^{∞} and E'^{∞} respectively the asymptotical value of $deps(\mathsf{a})$ at p and q.

For starters, we show that p and q agree on aborted commands. Assume that (say) p aborts a in λ, that is $E = \mathit{abort}$. Applying Proposition 2, $\mathsf{DDS}.\mathit{announce}(\mathsf{a})$ returns (abort, f) in λ. Similarly, from the fact that q decides value E' for $\mathit{deps}(\mathsf{a})$, $\mathsf{DDS}.\mathit{announce}(\mathsf{a})$ returns (E', f') in λ. As $E \notin 2^{\mathscr{C}}$, the Weak Agreement property of DDS implies that $f = f' = \mathit{false}$. According to the definition of check_f, $\mathit{check}_{\mathsf{CONS_b}}(E, E)$ and $\mathit{check}_{\mathsf{CONS_a}}(E', E')$ are *true* at respectively p and q. Since $\mathsf{CONS_a}$ is verifiable, $\mathit{check}_{\mathsf{CONS_a}}(E, E)$ implies that $\mathsf{CONS_a}$ returns E in λ. Similarly, we know that $\mathsf{CONS_a}$ returns E' in λ. From the Agreement property of consensus, $E = E' = \mathit{abort}$.

The above reasoning tells us that Stability holds if a is aborted at any correct process. Now, pick $\mathsf{b} \in E^{\infty} \setminus E'^{\infty}$.

- (Case b was aborted at q.) Necessarily b is aborted at p. Contradiction.
- (Case b is not in E'.) Applying Proposition 2, $\mathsf{DDS}.\mathit{announce}(\mathsf{a})$ returns both (E, f) and (E', f') in λ. We follow the same case analysis as in the crash-prone case—given in the proof of Theorem 2. Namely,
  - (Case $f = \mathit{true}$.) In that case, by the Weak Agreement property, $E = E'$. Contradiction.
  - (Case $f' = \mathit{true}$.) Symmetrical to the previous one.
  - (Case $f = f' = \mathit{false}$.) According to the definition of check_f, $\mathit{check}_{\mathsf{CONS_b}}(E, E)$ and $\mathit{check}_{\mathsf{CONS_a}}(E', E')$ are *true* at respectively p and q. Since $\mathsf{CONS_a}$ is verifiable, $\mathit{check}_{\mathsf{CONS_b}}(E, E)$ implies that $\mathsf{CONS_a}$ returns E in λ. Similarly, $\mathsf{CONS_b}$ returns E' in λ. By the Agreement property of consensus, $E = E'$ Contradiction.

(Reliability) It remains to prove that once submitted, a command c always ends-up being decided at every correct process q. In other words, that the protocol attains the Reliability property defined in §3.2.4. A client is correct and upon submitting command c propagates it to $f + 1$ processes. Let p be a correct process among them. If p is the coordinator of c then it executes the *submit* handler immediately (line 1). Otherwise, either eventually c is decided at p, or its coordinator is considered as slow (line 2). Both CONS and DDS are wait-free services. It follows that the call at lines 3 and 4 return. Thus, p eventually executes line 5. Since links are reliable, process q eventually receives a tuple (c, D, b) from p (line 7). (D, b) is the value p returned from calling $\mathsf{DDS}.\mathit{announce}(c)$ (line 3). As DDS is trusted wrt. $(\mathit{id}_{\mathscr{C}}, \_)$, $\mathit{check}_{\mathsf{DDS}}(c, D, b)$ holds. Similarly, as CONS_c is trusted wrt. (c, DDS), $\mathit{check}_{\mathsf{CONS}_c}(D, D)$ is true. Thus, the verifier $\mathit{check}_f((c, D, b), D)$

returns *true* and the assignment at line 9 occurs. It follows that c is eventually decided at p, as required.

□

4.3 Implementations

In this section, we present several BLSMR protocols, each focusing on the optimization of a specific metric to implement Byzantine state-machine replication.

4.3.1 Preliminaries

In what follows, we assume that each process p holds a *log* that stores the commands it has received so far. This object provides a unique operation *conflicts*(c). It adds c to the log then returns all the commands b present in the log and conflicting with c (i.e., $\mathsf{b} \asymp \mathsf{c}$).

As usual [41], we assume that correct recipients of a message that carries an incorrect signature do not process it. Similarly, we consider that commands and dependencies are signed and that signatures are verified (§2.2.1).

About Progress

Algorithm 3 assumes a wait-free implementation of the DDS service. This means that each call to *announce*(c) made by a correct process eventually returns. Achieving this despite failures and asynchrony, while offering low complexity, requires to retry transmitting command c to a quorum of replicas if the command was stucked previously. Hereafter, we detail instead best-effort implementations of the DDS service. This means that such constructions satisfy the following property:

Best effort If a correct process calls DDS.*announce*(c) infinitely often then the service eventually returns.

Transforming a best-effort implementation into a wait-free one is fairly simple. The process repetitively calls the service until it answers. Once this occurs, the response value is returned to the upper layer.

Stronger Byzantine Quorum System

For reasons that will be clear in later sections, we propose a special case of masking Byzantine Quorum System [101], called *Strong Byzantine Quorum System* (S-BQS). To this end, recall that $\mathscr{Q}$ denotes the quorum system and $\mathscr{B}$ all the possible sets of faulty Byzantine processes. S-BQS is defined in terms of the properties below.

S-Consistency $\forall Q_1, Q_2 \in \mathscr{Q}. \forall B_1, B_2, B_3 \in \mathscr{B} : (Q_1 \cap Q_2) \setminus (B_1 \cup B_2) \not\subseteq B_3$

S-Availability $\forall B \in \mathscr{B}. \exists Q \in \mathscr{Q} : B \cap Q = \varnothing$

S-Consistency guarantees that any two quorums intersect in $3f+1$ processes and thus in $2f+1$ correct ones. S-Availability ensures that there exists a quorum containing only correct nodes.

4.3.2 À la EPaxos

Using our framework to create a Byzantine-hardened LSMR protocol boils down to providing concrete implementations of the services described in Algorithm 3 and showing that they abide by the expected properties, e.g., (Byzantine) agreement for consensus. In this section, we do such an exercise for the EPaxos protocol [106].

For simplicity, we consider a variation of EPaxos which does not involve sequence numbers and without fast path. We discuss the impact of adding a fast path later in the section. We begin by presenting and explaining the DDS service of EPaxos in the Byzantine case.

Byzantine DDS

Algorithm 4 depicts a Byzantine best-effort DDS service. Internally Algorithm 4 relies on a service which returns for a given command a Byzantine quorum (variable *BQS*). *BQS* can be implemented using any kind of dissemination quorum system (DQS for short) presented in §2.2.3. For instance, quorums may simply be sets of $2f+1$ processes among $3f+1$.

Using a DQS is important because when queried a Byzantine processes may lie, omitting certain (or all) dependencies of a command. The D-Consistency property of DQS guarantees that any two quorums have at least one correct process in common. Indeed, this property requires that the intersection between any two quorums is of size $f+1$. Thus, at least one process behaves correctly

Algorithm 4 Acquiring dependencies of a command c à la EPaxos – code at process p

```
1:  announce(c) :=
2:      eff:  Q ←(BQS,c)→ BQS.getQuorum(c)
3:            broadcast(c) to Q
4:            wait until recv(c, deps_j(c)) from all j ∈ Q
5:            D ← ⋃_{j∈Q} deps_j(c)
6:            D.Γ ← (c, (deps_j(c))_{j∈Q}, Q)
7:            return (D, false)
8:
9:  when recv(c) from p
10:     pre:  check_{id,℘}(c, true)
11:     eff:  deps(c) ← log.conflicts(c)
12:           send(c, deps(c)) to p
13:
14: check_DDS(c, (D, b)) :=
15:     f ← ∧ check_BQS(c, D.Γ.Q)
16:         ∧ b = false
17:         ∧ |D.Γ.deps| = |D.Γ.Q|
18:         ∧ D = ⋃_{d∈D.Γ.deps} d
19:     return f
```

and reports all the conflicting commands it has seen so far, ensuring that two conflicting commands see each other (the Visibility property of LSMR).

In detail, *BQS* is a verifiable wait-free service that provides a single operation *getQuorum*. For a given quorum system $\mathcal{Q}$ satisfying the DQS properties, *getQuorum*(c) returns some element of $\mathcal{Q}$. The service ensures (**Exhaustivity**) for each quorum $Q \in \mathcal{Q}$, if called infinitely often with input c, *BQS* eventually returns Q. Such a property can be implemented easily using a seed s and a VRF f (see §2.2.3). The seed s is included in $Q.\Gamma$. This allows running $check_{BQS}(\mathsf{c}, Q)$ to verify that Q is the actual output of f for c and s.

To announce a command c, at line 3 in Algorithm 4, the coordinator broadcasts it to a quorum obtained through *BQS* (at line 2). When a process in the quorum receives the command c (at line 9) it computes (and stores) the set of commands conflicting with c it has seen so far (line 11). The reply is sent at line 12. Note that the process returns a tuple with the command and its set of dependencies signed (implicit as we have mentioned in §4.3.1).

After receiving the replies from all the processes in quorum Q, process p calculates the union of all the dependencies it received (line 5). Then, p generates and embeds the proof Γ (at line 6), necessary to make the service verifiable. The proof Γ needs to provide enough evidences so that other processes can confirm that the process in charge of the announce call did not misbehave during its actions. The algorithm returns the tuple (D,*false*) at line 7. The hardcoded value *false* indicates that there is no fast path in this implementation.

Function $check_{\mathsf{DDS}}(\mathsf{c}, (D, b))$ defined at line 15 ensures that Algorithm 4 is a trusted service. In particular, it makes sure that it is verifying wrt. $(id_{\mathscr{C}}, \_)$, only processing commands actually submitted by clients. Notice that for efficiency purposes, it would be possible to trim commands that are not conflicting with c at line 5. This would require in $check_{\mathsf{DDS}}(\mathsf{c}, (D, b))$ to also do such a verification. This improvement is omitted for clarity.

▷ Theorem 6. Algorithm 4 implements a best-effort trusted DDS service.

Proof. Consider some run λ of Algorithm 4.

(Visibility) Let a, b be two conflicting commands. Assume D, D' are respectively the return values to $announce_p(\mathsf{a})$ and $announce_q(\mathsf{b})$ during λ. These two values are returned at the correct process p and q, respectively.

The set of dependencies of a command is returned at line 7, yet calculated at line 5 through the execution of the union operator. Consider the execution of this line by process p. The union operator is parameterized with a quorum Q_1, obtained through the execution of line 2. Let Q_2 be the quorum obtained by process q when it executes line 2. By the D-Consistency property of the DQS service the quorums Q_1 and Q_2 intersect in at least $f+1$ correct processes and thus, at least 1 correct process executed line 11 and line 12 . This implies that it exists at least a correct process where $\mathsf{b} \in deps(\mathsf{a})$ and therefore $\mathsf{b} \in D$. An analogous reasoning can be used for process q, there will be at least 1 correct process that will report $\mathsf{a} \in deps(\mathsf{b})$ at line 11 and send it at line 12 and thus $\mathsf{a} \in D'$ at line 5.

(Weak agreement) Trivial as the service returns (_,*false*) at a correct process.

(*Trust*) Next, we explain how $check_{\mathsf{DDS}}$ ensures that Algorithm 4 performs correctly on the input c. This function consists of a series of safety verifications in order to bound the damage a Byzantine coordinator would made on the service. At line 15, we first check that Q is a proper quorum. Further, at line 16, we verify that there is no fast path. At line 18 the function recalculates the union of dependencies to guarantee that the coordinator did

not omit anything. If all of the above checks are passed, the function returns *true*. Finally, as a correct replica verifies that c is an actual command at line 10, this ensures that Algorithm 4 is verifying wrt. $(id_{\mathscr{C}}, \_)$.

(Best-effort) Consider that a correct process p calls infinitely often *announce*(c). At line 2, p returns a Byzantine quorum from the verifiable service *BQS*. This service ensures the Exhaustivity property, that is for each quorum $Q \in \mathscr{Q}$, if called infinitely often with input c, *BQS* eventually returns Q. Let B be the failure pattern of λ. By the D-Availability property of $\mathscr{Q}$, there exists a Q_0 such that $B \cap Q_0 = \emptyset$. Consider the first time t at which p retrieves Q_0 from *BQS* at line 2. As *BQS* is wait-free the assignment at line 2 happens. Thus p broadcasts a message (c) to all the processes in Q_0. As links are correct and these processes are not in B, they eventually deliver this message and each returns a set of dependencies to p. Thus, there exists a time $t' > t$ after which line 4 holds. It follows that p returns a value at line 7.

□

Consensus

To finish building BLSMR à la EPaxos, we need $\mathsf{CONS_c}$ to be a trusted service wrt. $(\mathsf{DDS}, \mathsf{c})$. As mentioned earlier in §4.2.2, to achieve this we can use any BFT SMR protocol that provides a primitive to agree on a value. For our use case, we consider HotStuff [139] and expect it to be modified such that correct processes decide iff $check_{\mathsf{DDS}}(\mathsf{c}, (D, b))$ holds.

To make the service verifiable, we also need a function $check_{\mathsf{CONS_c}}(D, D)$. This function is implemented by using the COMMIT certificates (see §2.3.4) generated by HotStuff: the COMMIT certificates are the proof Γ and function $check_{\mathsf{CONS_c}}(D, D)$ verifies their correctness exactly as detailed in [139]. Finally, we require the existence of a procedure to select $3f + 1$ replicas in Π to run $\mathsf{CONS_c}$. This construction follows the one we used for the *BQS* service in Algorithm 4.

Adding a fast path

In Algorithm 4, we are always returning *false* for the flag that indicates if the fast path is taken. If we want to add a fast path to the service, we need to consider bigger intersection sizes in the BQS service. More precisely, we need a strong

Byzantine quorum system (introduced earlier in §4.3.1), that is an intersection size of $3f + 1$ across quorums.

To understand why this is required, consider that the fast path was taken for some command c and that its initial coordinator is slow, therefore a recovery is triggered. The new coordinator queries the initial quorum used for command c (say, Q_c) and waits for $|Q_c| - f$ replies. Concurrently, an analogous recovery procedure is taking place for command d, conflicting with c, with quorum Q_d. The coordinator also awaits $|Q_d| - f$ replies. In this scenario, we basically ran out of replicas to maintain the consistency property of BLSMR. Hence, the fast path requires an extra $2f$ replicas in the intersection between quorums.

If we consider that we have such an intersection property necessary, we can then add a fast path to Algorithm 4 using the usual mechanism of EPaxos, that is dependencies must match. In this case, we may also provide the following refinement to make the verification procedure in $check_{\mathsf{DDS}}$ more lightweight: If a coordinator claims that the fast path was taken, a process needs to verify that such a criterion is met. For performance, this can be implemented using threshold signatures (§2.2.3). In detail, at line 12 in Algorithm 4, we require that a process applies an extra signature (in this case a partial one) to $(\mathsf{c}, deps(\mathsf{c}))$. In this way, once a coordinator receives all the replies (each one partially signed) and verifies that they are all equal (i.e. the fast path condition is met), it signs the resulting message with a combined signature stored in Γ at line 6. This signature is then checked appropriately in the verifier $check_{\mathsf{DDS}}$.

4.3.3 Wintermute

We present now an instantiation of the services such that the resulting protocol has better load and message complexity than common BFT SMR protocols. The protocol can be seen as an improved and more practical version of Algorithm 4. We call this protocol *Wintermute* and begin its construction by addressing a well-known problem that has implications on the underlying abstractions.

4.3.4 Efficiently representing dependencies

In Algorithm 4, dependencies monotonically grow over the course of execution. To be practical, protocols must compact them, over-approximating the commands included in each set if required. Such a problem is common to all generic and leaderless SMR protocols (e.g., [90, 112, 106, 62]. For instance, without compaction, c-structs in Generalized Paxos can get arbitrarily large during a ballot. In what follows, we examine this problem and present various solutions.

Checkpointing

A first solution consists in the use of *checkpoints*. In the case of Generalized Paxos, such an approach is detailed in one of our previous works [117]. The key idea is that commands that are stable at a quorum of replicas are trimmed from the dependencies (or c-structs). Trimming occurs when a special checkpoint command that conflicts with everything else is learned.

Checkpointing has been used in practice previously. In particular, the authors of [127] report that they need to checkpoint every few thousands of commands to keep the system stable. As a consequence, such a technique has clear limitations at large scale. Therefore, to solve this issue we instead consider a compact representation of the dependencies, as presented next.

Compaction

Upon submission, each command c gets assigned a sequence number, or *timestamp*, denoted c.*ts*. Dependencies are represented as a vector of timestamps. For instance, consider the example below, where the clients are $\{a, b, c\}$, using Algorithm 4 for reference. A compacted representation of a *deps*(c) received from some process j in the quorum at line 4 could be the following:

$$deps_j(\mathsf{c}) = \{a \mapsto 2, b \mapsto 7, c \mapsto 4\}$$

This is interpreted as follows: command c depends on all the commands of client a up to timestamp 2, up to timestamp 7 for client b and up to timestamp 4 for client c. Note that such a representation is an over-approximation. Indeed, it might be the case that the command with timestamp 1 by a is in fact commuting with c. The representation is made exact when using *version vectors with exceptions*, an approach detailed in [103].

The above technique has a few issues. First, in a system where clients can enter and leave the system, the dimensions of the vector can grow arbitrarily. To solve this problem, one may again resort to the use of checkpoint and a special epoch entry. Once executed, a checkpoint clear the vector from all its entries except the epoch one that is incremented.

The second issue is that even if one assumes that clients are always correct there is still the possibility that a Byzantine replica once queried for the dependencies of a command reports back false dependencies by faking timestamps when calculating conflicts (Algorithm 4 at line 11). A correct process once receiving such dependencies would have to spend time checking which entries are

signed and the ones that are not (fake ones crafted by a malicious process). We provide a solution to this problem in §4.3.5.

Further compaction

For performance, we aim at further compacting dependencies, using instead vectors of the size of the replica set. Extra care must be taken in this case, as Byzantine replicas can exhaust timestamps on purpose, or assign twice the same timestamp. This problem also occurs in a system where clients are Byzantine.

To deal with these issues we propose that timestamps are generated by a verifiable timestamping service, TS. Given some command c, this service offers a single operation $\mathit{next}(\mathsf{c})$ returning a timestamp, that is an element t from some totally ordered set $(\mathcal{T}, <)$. It ensures the following properties:

Uniqueness If TS returns t and t' then $t \neq t'$.

Monotonicity If TS returns t then t', necessarily $t < t'$.

Gap-free If TS returns t and immediately after t', then there is no t'' such that $t < t'' < t'$.

Below, we present two possible implementations of the timestamping service, one relying on dedicated hardware and the other being distributed.

- To implement a verifiable timestamping service, we can use trusted hardware, following the mechanism in [134]. Here, the authors present a non-interactive solution based on a service they name *Unique Sequential Identifier Generator* (USIG). This service is capable of generating unique, monotonic and sequential (thus gap-free) identifiers.

- Another mechanism is to use the non-skipping timestamps proposed in [19]. Based on it, we propose the following instantiation for TS: For each replica p, we define a set of $4f + 1$ processes in charge of maintaining the service. To this end, they use a clock variable initially equal to 0. When executing $\mathit{next}(\mathsf{c})$, p queries the processes and await for $3f + 1$ replies containing the clock value of the service processes. Process p then takes the $(f + 1)^{st}$ largest timestamp from the replies and adds 1 to it. The resulting timestamp $\hat{t}$ is the output of the TS service. To make the service verifiable we store in $\hat{t}.\Gamma$ the replies received. The implementation of function $\mathit{check}_{\mathsf{TS}}$ is trivial: one needs to require that replied the timestamps are signed and then re-calculate the $(f + 1)^{st}$ largest timestamp from them.

To guarantee the properties of the service, before returning $\hat{t}$, process p sends it to the service processes. Upon receiving such a message, a service process updates its clock to the maximum value between $\hat{t}$ and the local clock, then acknowledges p. Process p returns once it has received $3f + 1$ such acknowledgments. Notice that this last phase can be piggyback on the first phase of the previous inquiry to the TS service, saving a message round-trip. A proof of correctness of the schema is available in [19].

The above two implementation relies both on integers. In practice, it may overflow leading to common problems [138]. Some prior works investigate the question of bounding timestamps [82]. However to the best of our knowledge, there is no existing solution in the case of Byzantine systems.

Before closing this section, we note that when the representation of dependencies is of the size of the replica set, two key problems must be solved. First, it is not possible to over-approximate the dependencies and version vectors with exceptions are necessary [103]. Second, due to recovery, a command might get assigned different timestamps by different coordinators. In that case, its timestamp is simply the union of these values (as proposed in [130]). To ease presentation, and avoid such subtleties, the description of Wintermute that follows assumes that timestamp are assigned by clients and that dependencies are of the size of the number of clients.

4.3.5 Protecting the dependencies representation

When dependencies are compacted they need to be protected. Indeed, a Byzantine replica may simply report non existing commands as dependencies. To solve this problem, a first solution is to sign each entry in the vector and check them where necessary. This approach is however expensive, as lines 5 and 18 in Algorithm 4 now costs $O(n^2)$ verifications, where n is the size of the replica set. We propose instead the use of the special Strong Masking Byzantine Quorum System (S-BQS) defined in §4.3.1 in conjunction with a technique called threshold union, initially introduced in our previous work on the Atlas protocol [62].

Threshold union

By the S-Consistency property of the S-BQS we know that $2f+1$ correct processes exist in the intersection between any two quorums. Thus, when calculating the union of dependencies received from a quorum we are safe from the attack mentioned above by including only commands that were reported by at least $f + 1$ processes. This is the very purpose of threshold union operator.

Formally, consider a command c and a quorum Q obtained through a call to some quorum system service. The through union operator is defined as follows:

$$\bigcup_{j\in Q}^{f+1} deps_j(\mathsf{c}) = \{\mathsf{c} \mid count(\mathsf{c}) \geq f+1\}$$
$$count(\mathsf{c}) = |\{j \in Q \mid \mathsf{c} \in deps_j\}|.$$

To illustrate the above equation, consider that the set of clients consists of $\{a,\ldots,d\}$, and that $f=1$. The following (compacted) dependencies are received from three replicas in the DQS quorum Q (e.g., line 4 in Algorithm 4).

$$deps(\mathsf{c}) = \{a \mapsto 2, b \mapsto 7, c \mapsto 0, d \mapsto 0\}$$
$$deps(\mathsf{c}) = \{a \mapsto 1, b \mapsto 4, c \mapsto 0, d \mapsto 0\}$$
$$deps(\mathsf{c}) = \{a \mapsto 1, b \mapsto 99, c \mapsto 0, d \mapsto 0\}$$

The result after applying the threshold operator is as follows:

$$D = \{(a,0),(a,1),(b,0),\ldots,(b,7)\}$$

Due to compaction, a replica might know that d depends on c, yet without knowing d itself. The threshold union operator ensures that if $d \in deps(c)$, then necessarily d is known at $f+1$ replicas. Thus, if a command is unknown at some replica, it can be fetched remotely without having to wait recovery.

Building the quorum system

In what follows, we explain how we construct the S-BQS quorum system. We also analyze its load and availability.

Our Byzantine quorum system is built as a variation of the *M-grid quorum* system proposed in [102]. Let us assume $|\Pi| = n$, $\mathscr{B} = \{B \subseteq \Pi : |B| = f\}$ and $f \leq (\sqrt{n}-2)/3$. We arrange the set of replicas into a $\sqrt{n}\times\sqrt{n}$ grid. Let $(R_i)_i$ and $(C_j)_j$ be the rows and columns of the grid, respectively, with $1 \leq i,j \leq \sqrt{n}$. The key idea is that a quorum consists in ζ rows and ζ columns, with $\zeta = \lceil \sqrt{3f/2+1} \rceil$. In other words, the quorum system is defined as:

$$\mathscr{Q} = \left\{ \bigcup_{j\in J} C_j \cup \bigcup_{i\in I} R_i : J,I \subseteq \{1,\ldots\sqrt{n}\}, |J| = |I| = \zeta \right\}$$

We illustrate this construction in Figure 4.1, with $n = 9^2$ and $f = 1$. In this figure, as $\zeta = 2$, a quorum consists of two rows and two columns.

▷ Theorem 7. $\mathscr{Q}$ is a Strong Byzantine Quorum system for $f \leq (\sqrt{n}-2)/3$.

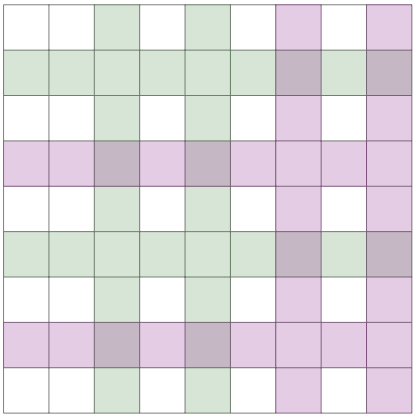


Figure 4.1. Example of quorums in the S-BQS quorum system, with $f \leq 2$, $n = 9^2$ and thus $\zeta = 2$. Each quorum (in green and purple) consists of two rows and two columns. Their intersection is shaded.

Proof. Consider $Q_1, Q_2 \in \mathcal{Q}$. We first establish that S-Consistency holds, that is every intersection contains at least $3f + 1$ processes. If Q_1 and Q_2 have in common either a column or row, then $|Q_1 \cap Q_2| \geq \sqrt{n} \geq 3f + 2$. Otherwise, $[columns(Q_2) \cap rows(Q_1)] \cap [columns(Q_1) \cap (rows(Q_2)] = \varnothing$. In such a case, the ζ columns of Q_1 intersect once with the ζ rows of Q_2; and vice-versa. It follows that $|Q_1 \cap Q_2| \geq 2\zeta^2 > 3f + 1$. To establish S-Availability, observe that $\sqrt{n} - f \geq 3f + 2 - f \geq 2f \geq \zeta$. As a consequence, for any $B \subseteq \Pi$ such that $|B| = f$, we may pick ζ rows and ζ columns whose union does not contain any element in B. □

▷ Theorem 8. The load of $\mathcal{Q}$ belongs to $O(\sqrt{f/n})$ when quorums are picked uniformly at random.

Proof. Let $c(\mathcal{Q})$ be the size of the smallest quorum in the S-BQS quorum system $\mathcal{Q}$. From [101], we know that the load of $\mathcal{Q}$ is $c(\mathcal{Q})/n$. All the quorums in $\mathcal{Q}$ system are of the same size, that is $2 \times \zeta \times \sqrt{n} - \zeta^2$. Hence, the load of $\mathcal{Q}$ is given by: $(2 \times \zeta \times \sqrt{n} - \zeta^2)/n \in O(\sqrt{f/n})$ □

Discovering dependencies

Algorithm 5 depicts the DDS service of Wintermute. Its structure is similar to Algorithm 4. The key differences are highlighted in blue. They consist in the compact representation of dependencies, the use of a S-BQS quorum system instead of a DQS one and the application of the threshold union operator to trim

potentially misleading dependencies from Byzantine replicas. The correctness of this construction is established below.

Algorithm 5 Wintermute – Acquiring dependencies of a command c – code at process p

```
1:  announce(c) :=
2:      eff:  Q ←^{BQS,c} BQS.getQuorum(c)
3:            broadcast(c) to Q
4:            wait until recv(c, deps_j(c)) from all j ∈ Q
5:            D ← ⋃^{f+1}_{j∈Q} deps_j(c)
6:            D.Γ ← (c, (deps_j(c))_{j∈Q}, Q)
7:            return (D, false)
8:
9:  when recv(c, t) from p
10:     pre:  check_{id_𝒞}(c, true)
11:     eff:  deps(c) ← log_p.conflicts(c)
12:           send(c, deps(c)) to p
13:
14: check_DDS(c, (D, b)) :=
15:     f ← ∧ check_BQS(c, D.Γ.Q)
16:         ∧ b = false
17:         ∧ |D.Γ.deps| = |D.Γ.Q|
18:         ∧ D = ⋃^{f+1}_{d∈D.Γ.deps} d
19:     return f
```

▷ Theorem 9. Algorithm 5 implements a best-effort trusted DDS service.

Proof. Consider some run λ of Algorithm 5.

(Visibility) Let a and b be two conflicting commands. Assume that two correct processes p and q return $(D, \_)$ and $(D', \_)$ from calling respectively *announce*(a) and *announce*(b) during λ.

The set of dependencies of a command is returned at line 7, but calculated at line 5 through the execution of the threshold union operator. Consider the execution of this line by process p. The threshold union operator is parameterized with a quorum Q_a, obtained through the execution of line 2.

Similarly, let Q_{b} be the quorum obtained by process q when it executes line 2.

By the S-Consistency property of the S-BQS service the quorums Q_{a} and Q_{b} intersect in at least $2f + 1$ correct processes. These processes all execute lines 9 to 12. The handler is atomic. Hence, either $f + 1$ such processes execute these lines for a before b, or vice-versa. Without lack of generality, consider that this is the later case that happens during λ. For each of these processes j, $\mathsf{b} \in \mathit{deps}_j(\mathsf{a})$ holds. Hence, b is reported $f + 1$ times in $(\mathit{deps}_j)_{j \in Q_{\mathsf{a}}}$. Thus, applying the definition of the threshold union operator, we have $\mathsf{b} \in D$, as required.

(Weak agreement) Immediate as a call to *announce* always return $(\_, \mathit{false})$ at a correct process.

(Trust) This is analogous to the proof provided in Theorem 6. The main difference is that now at line 19, the verifier calculates the threshold union of dependencies instead of base union.

(Best-effort) The proof is similar to the one above for Theorem 6. It relies on the S-Availability property of the S-BQS quorum system (instead of the D-Availability of the DQS quorum system earlier).

□

Consensus

As mentioned in §4.1.1, a typical deployment of Wintermute runs $\mathsf{CONS}_{\mathsf{c}}$ over a few nodes, whereas order of magnitude more are running the DDS service. As a consequence, for the consensus service, we make use of a protocol that favors high performance (that is, low latency and low authenticators complexity) over scalability. Namely,, we use the *Quorum* algorithm [17], to drive the decisions in $\mathsf{CONS}_{\mathsf{c}}$. This protocol is modified so that correct processes decide over D only if $\mathit{check}_{\mathsf{DDS}}(\mathsf{c}, (D, b))$ returns true.

To lower the amount of signatures sent to all processes in Algorithm 3 at line 5, we modify *Quorum* to work with threshold signatures in the following way: The coordinator sends D to a pseudo-randomly selected set of $3f + 1$ replicas (again using VRFs). The replicas apply a partial signature to this message and reply. If the coordinator in $\mathsf{CONS}_{\mathsf{c}}$ receives $2f + 1$ matching replies where all replies are properly partially signed, then D is committed. Otherwise, D is aborted and PBFT is run (as explained in §2.2.6). Once in possession of the

matching replies, the coordinator proceeds to combine the signatures received into a single threshold signature on D (this will be embed as $D.\Gamma$ to be used in $check_{\mathsf{CONS}}(D, D)$) and broadcast the result to Π. Each process in Π receives only D with a single threshold signature after this step. As a consequence, the authenticators complexity of Wintermute is bounded by the DDS service. We detail this point in the next section.

The above modification makes sense because we do not have a fast path. The processes in CONS_c only decide if the expensive call to $check_{\mathsf{DDS}}(c, (D, b))$ is evaluated to true. Other processes in Π, once receiving D, just need to evaluate $check_{\mathsf{CONS}}(D, D)$.

4.4 Complexity Analysis

Wintermute is the result of instantiating our framework with the following services: *(i)* the timestamping service TS (§4.3.5), *(ii)* the S-BQS quorum system built via the M-grid approach presented in §4.3.5, *(iii)* the *Quorum* consensus algorithm [17] as CONS. We consider that the processes in a quorum of the *BQS* and processes in CONS are pseudo-randomly and uniformly selected via VRFs. In what follows, we explain the values reported in Table 4.1 regarding the complexity of the Wintermute protocol. As motivated in §4.1.1, we are under the assumption that f is small over n ($f \ll n$).

Latency. The latency of the DDS service in the critical path is of 4 message delays. The first 2 come from the use of the TS service and the other 2 from the execution of lines 3 and 4 in Algorithm 5. As mentioned earlier *Quorum* is used to instantiate CONS, which in the critical path gives us an extra 2 message delays. Thus, the total latency (Δ) of Wintermute in the critical path is 6.

Message Complexity. Since *BQS* is instantiated as S-BQS via the M-grid approach, the size of a quorum has a complexity in $O(\sqrt{fn})$. In Algorithm 5, a process p sends a single message (line 3) exclusively to $|Q|$ processes and at line 12 a process replies exclusively to p. Hence, we have a message complexity μ of $O(\sqrt{fn})$.

Authenticator Complexity. Wintermute's authenticator complexity is bound by the DDS service. In Algorithm 5 at line 4 a process p receives $|Q|$ signed messages from a quorum Q. This is also the maximal amount of signatures a process receives later on in the algorithm. Hence, we have α belongs to $O(\sqrt{fn})$.

Load. In the case of Wintermute, the load of the protocol is the sum of the probabilities of a process being picked in the DDS service or CONS service. Under the assumption $f \ll m$, the load of CONS is negligible. Thus, the load of the

protocol is bound exclusively by the load of the DDS, which is $O(\sqrt{f/n})$ (that is, the load of the *BQS* service).

4.4.1 Possible Extensions

Leveraging the modular nature of our framework allows one to propose different instantiations of Wintermute that use techniques that have recently been explored in BFT replication.

Fast Path. As with EPaxos (§4.3.2) adding a fast path in Wintermute is possible. However, this requires an additional $2f$ replicas in the intersection between quorums (for a total of $5f + 1$ replicas).

Trusted Hardware. As mentioned in §4.3.4, we can use trusted hardware in the vein of [134] to implement the TS service. This lower the latency of the protocol, saving two message delays in the critical path.

Tree Aggregation. Further development can be done to implement DDS using a tree topology with dependencies being gathered along the tree. A similar approach is taken in [108] where the authors provide an implementation of HotStuff using trees, which avoids the one-to-all communication pattern of the original protocol.

Gossip. As Wintermute is aimed towards scenario where the system has hundreds to a thousand of replicas, relying on a base broadcast protocol at that scale is not feasible. Tackling the issue by relying on propagating messages through gossiping is a possible solution and is the approach taken in some modern BFT protocols like Gosig [95] and Tendermint [34]. We believe that such an adaptation is possible for our protocol—that is using a gossip protocol to implement the broadcast abstraction in Algorithm 3 and Algorithm 5.

4.5 Related Work

As mentioned earlier, the work of Cachin et al. [37] formalizes the idea of a *validated* protocol and proposes the *external validity* property for Multi-Valued Byzantine Agreement. Optimizing the communication cost in both bits and message delays of Multi-valued Byzantine Agreement is further studied in [99, 5] and can be of interest to analyse the cost of the proofs our trusted services rely on.

The *Verifiable and Provable Consensus* abstraction inspired by [37] and presented in [125] is used in BFT-SMaRt [24]. Later on, the work in [126] describes how BFT-SMaRt is used in the permissioned blockchain platform Hyperledger

[12]. In this chapter we have mainly focused on scaling the underlying protocol to be used in a permissioned blockchain. From a practitioners point of view the work in [23] addresses other challenging subtleties (e.g. decentralized reconfiguration) that one might encounter during real implementations of a blockchain.

Modern BFT. The revamped interest in BFT replication and its application to blockchains has led researchers to revisit protocols that have asymptotic behavior similar to PBFT (for instance, this is the case with BFT-SMaRt) to develop improved versions that offer better scalability features such as non-quadratic message complexity and lower authenticators complexity. This is the case with HotStuff [139] and SBFT [74].

In terms of message delay, new protocols try to avoid the three-round latency of PBFT to commit a value in the good-case (when the leader is honest but the other replicas may be faulty and the network is synchronous). This the case with the work in [6] where the authors propose a two-round authenticated BFT SMR protocol that requires $n \geq 5f-1$. Interestingly, the recent work in [86] show that $5f-1$ is the tight lower bound for fast Byzantine consensus, that is, the (optimal) two-round latency comes with the cost of lower resilience. Such bound should also apply to Byzantine leaderless protocols.

Other modern approaches to scale BFT replication that have been recently studied are related to gossip-based message propagation, collective signing [129], sharding [119] (i.e. partial replication) and exploiting the topology of the network [39]. Gossip-based consensus protocols have been recently studied [40] to confront the challenges faced by SMR in large geographically distributed systems. In the context of permissioned blockchains a notable example of such approach is Tendermint [34]. A tree communication strategy is adopted by ByzCoin [85] which employs such a communication tree in combination with collective signatures. Recently, similar approach is taken in [108], where the authors propose a BFT communication abstraction that organizes participants in a tree which allows vote aggregation to be performed. Most protocols though rely on a combination of techniques, as for example Gosig [95] that relies on gossip propagation, VRFs for proposer election (as in Algorand [70]) and a multi-signature schema.

Leaderless. Byblos [20] is based on the non-skipping timestamps technique [19], requires $n > 4f + 1$ replicas and has $O(n^2)$ message complexity. More recently, the protocol ISOS [60] was proposed and can be seen as Byzantine version of EPaxos [60], it includes a fast path but has a $O(n^2)$ message complexity. In terms of application to permissioned blockchains a notable byzantine leaderless protocol is DBFT [46], used to power the RedBelly [47] blockchain. The work in [118] present a family of leaderless byzantine consensus protocols that seem to be scalable but are aimed at permissionless blockchains, thus the resulting

protocols offer probabilistic guarantees.

Chapter 5

Conclusion

5.1 Summary

In this thesis we proposed a framework that captures the essence of leaderless state-machine replication (Leaderless SMR). We introduced a set of desirable properties for Leaderless SMR: (R)eliability, (O)ptimal (L)atency and (L)oad balancing, and we showed that protocols matching all of the ROLL properties are subject to a trade-off between performance and reliability. We established a lower bound on the message delay to execute a command in protocols optimal for the ROLL properties. This lower bound explains the persistent chaining effect observed in experimental results. Further, we extended our framework to support Byzantine failures and provided an instantiation of it that can be used to implement more scalable permissioned blockchains.

5.2 Future Work

In relation to the content of Chapter 3, a journal version of [116] is planned and will address questions regarding *liveness* more explicitly.

In terms of Chapter 4, we have an ongoing implementation[1] of Wintermute built upon the framework used in the evaluation of Tempo [61]. The framework allows for the prototyping of leaderless protocols to be created rather quickly. The variations we want to cover rely on the techniques mentioned earlier: gossip propagation, fast path, use of trusted hardware and different instantiations of Byzantine quorum systems. We envision an evaluation in the the style of [74] where one is able to run a real blockchain workload. Moreover, we conjecture

[1] Available at https://github.com/tuanir/fantoch/tree/byzantine

that such type of workload is mainly composed of highly commutative operations, which would strengthen the case for the use of BLSMR protocols. In a theoretical side, we plan on presenting an information theoretic protocol in the vein of Information Theoretic HotStuff [7], as well as a Byzantine version of Tempo [61].

Permissionless setting. Perhaps the most interesting and challenging continuation to our work lies in its adaptation to use in permissionless blockchains. The first idea we want to pursue and that would have less impact in the underlying abstractions is to bound the churn in the system via specially scheduled epochs in which new replicas can join. The second idea relies on exploring Byzantine quorum systems that are resilient to Sybil attacks (as in [124]) to build a DDS service resilient to such type of exploitation. Taking this route most likely requires to abstract what a failure means in the system. An example could be to build a byzantine quorum system in a proof-of-stake schema, where the intersection between quorums takes into account the economical powers of the adversary.

Appendix A

System Model

We formulate our results for an asynchronous distributed system augmented with failure detectors [42]. This section recalls the fundamentals of this common model of computation then present some technical lemmas. These lemmas are used in the follow-up to establish our complexity results regarding Leaderless SMR.

A.1 Model

We consider an asynchronous distributed system consisting of a finite set of *processes* $\Pi = \{p_1, p_2, \ldots, p_n\}$. Processes may fail-stop, or *crash*, and halt their computations. A failure pattern is a function $F : \mathbb{N} \to 2^{\Pi}$ that captures how processes crash over time. Processes that crash never recover from crashes, that is, for all time t, $F(t) \subseteq F(t+1)$. If a process fails, we shall say that it is *faulty*. Otherwise, if the process never fails, it is said *correct*.

Failure detectors. A failure detector is an oracle $\mathscr{D}$ that processes may query locally during an execution. This oracle abstracts information, regarding synchrony and failures, available to the processes. More precisely, a failure detector $\mathscr{D}$ is a mapping that assign to a failure pattern F, one or more histories $\mathscr{D}(F)$. Each history $H \in \mathscr{D}(F)$ defines for each process p in the system, the local information $H(p,t)$ obtained by querying $\mathscr{D}$ at time t. The co-domain of $H : \Pi \times \mathbb{N} \to R$ is named the range of the failure detector. An environment, denoted $\mathfrak{E}$, is a set of failure patterns.

In the vein in [50], we only consider *realistic* failure detectors. This class of failure detectors cannot forecast the future. This means that if two failure patterns F and F' are identical up to time t, then for any history $H \in \mathscr{D}(F)$, there exists $H' \in \mathscr{D}(F')$ identical up to time t to H.

Message buffer. Processes communicate with the help of messages taken from some set *Msg*. A message m is sent by some sender ($sender(m)$) and addressed to some recipient ($dst(m)$). The sender may define some content ($payload(m)$) before sending the message. A message buffer, denoted *buff* , contains all the messages that were sent but not yet received. More precisely, *buff* is a mapping from Π to 2^{Msg}. When a process p attempts to receive a message, it either removes some message from $buff[p]$, or returns a special null message. Note that p may receive the null message even if the message buffer contains a message m addressed to p.

Protocol. A protocol $\mathscr{P}$ consists of a family of n deterministic automata, one per process in Π. Computation proceeds in steps of these automata. At each step, a process p executes atomically one of the following instructions: receive some message m; fetch some value d from the local failure detector module; change its local state according to $\mathscr{P}$; or send some message m to another process. A *configuration* of algorithm $\mathscr{P}$ specifies the local state of each process as well as the messages in transit (variable *buff*). In some initial configuration of $\mathscr{P}$, no message is in transit and each process p is in some initial state as defined by $\mathscr{P}$.

Runs. A run of algorithm $\mathscr{P}$ using failure detector D in environment $\mathfrak{E}$ is a tuple $\lambda = (F, H, I, S, T)$ where F is a failure pattern in $\mathfrak{E}$, H is a failure detector history in $D(F)$, I is an initial configuration of $\mathscr{P}$, S is a sequence of steps of $\mathscr{P}$, and T is a growing sequence of times (intuitively, $T[i]$ is the time at which step $S[i]$ is taken). A run whose sequence of steps is finite (respectively, infinite) is called a finite (respectively, infinite) run. Every run λ must satisfy the following standard *(*well-formedness) conditions: *(i)* No process take steps after crashing; *(ii)* The sequences S and T are either both infinite, or they are both finite and have the same length; and *(iii)* The sequence of steps S taken in the run conforms to the algorithm $\mathscr{P}$, the timing T and the failure detector history H. A run λ is *admissible* for $\mathscr{P}$, or simply *is a run of* $\mathscr{P}$, when it is well-formed and in addition: *(fairness)* If λ is infinite, every correct process takes an infinite number of steps in λ. *(reliable links)* Every process that infinitely often retrieves a message from *buff* eventually receives every message addressed to it. We shall write $\mathscr{R}^{\mathscr{P}}$ the runs of algorithm $\mathscr{P}$. The superscript is omitted when the algorithm we refer to is unambiguous.

Our results mostly concern nice runs [81], that is failure-free runs during which the failure detector behave "perfectly". More specifically, we consider that a run is *nice* when there is no failure and the failure detector returns a constant value to the local process. $\mathscr{R}_n^{\mathscr{P}}$ denote the nice runs of algorithm $\mathscr{P}$.

Additional notations. When the context is clear, we do not distinguish a

run from its sequence of steps.[1] Below, we introduce a handful of operators and shorthands that leverage this simplification.

Consider two sequence of steps λ and λ'. We note $(\lambda|P)$ the sub-sequence of steps taken by the processes $P \subseteq \Pi$ in λ. Function $proc(\lambda)$ returns the processes that take steps in λ. We say that λ is *indistinguishable* from λ' to $P \subseteq \Pi$ when $\lambda|P = \lambda'|P$. As usual, if $\lambda \in \mathscr{R}$, λ' is indistinguishable from λ to Π and λ' is well-formed, then $\lambda' \in \mathscr{R}$. If $\lambda = \lambda[0]\ldots s\ldots\lambda[n]$, then $(\lambda| \leq s)$ is the sequence $\lambda[0]\ldots s$, and $(\lambda| \geq s)$ equals $s\ldots\lambda[n]$. The empty sequence is written ϵ. We note $\sqsubseteq$ and $\sqsupseteq$ respectively the prefix and suffix relations over the set of sequences.

Assume that an operation *op* is invoked in λ then later returns some response r to the local process. This corresponds respectively to the steps $inv(op)$ and $resp(op, r)$ in λ. We note $(\lambda|op)$ the sub-sequence of steps $(\lambda| \succeq inv(op)| \preceq resp(op, r))$, where $\preceq$ and $\succeq$ are respectively the reflexive closure of the happen-before relation ($\prec$) and the reflexive closure of the converse of $\prec$. For instance, $(\lambda|announce(c))$ refers to the steps taken in λ to announce command c.

A.2 Technical Lemmas

Below, we state a few results that concern nice runs of a protocol. As pointed out previously, in a nice run there is no failure and the failure detector always returns the same value. In this context, our first lemma is similar to Lemma 1 in FLP [64].

Lemma 3. *Consider two finite nice runs $\lambda\lambda'$ and $\lambda\lambda''$. If $proc(\lambda') \cap proc(\lambda'') = \emptyset$, then $\lambda\lambda'\lambda''$ is a nice run.*

Proof. Follow from the model definition. □

Lemma 4. *Consider a nice run $\lambda = \lambda' s s' \lambda''$. If $proc(s) \neq proc(s')$ and there is no message m such that $s = send(m)$ and $s' = recv(m)$, then $\hat{\lambda} = \lambda' s' s \lambda''$ is a nice run.*

Proof. First, we establish the well-formedness of $\hat{\lambda}$. Consider some receive step $recv(m)$ in $\hat{\lambda}$. As λ is well-formed, $send(m) <_\lambda recv(m)$. By hypothesis, $s \neq send(m)$, thus $send(m) <_{\hat{\lambda}} recv(m)$.

Then choose some process p. We have, $(\hat{\lambda}|p) = (\lambda'|p)(s's|p)(\lambda''|p)$, by distributivity of the projection operator. It remains to show that $(s's|p) = (ss'|p)$. There are three cases to consider: (Case $proc(s) = p$). As $proc(s) \neq proc(s')$, we have $(s's|p) = s = (ss'|p)$. (Case $proc(s') = p$). This case is symmetrical to the

[1] This is particularly true for a nice run, since the failure detector history is constant.

previous one. (Otherwise). We have $(s's|p) = \epsilon = (ss'|p)$. It follows that $\hat{\lambda}$ is indistinguishable from λ to Π.

Since $\hat{\lambda}$ is well-formed and $\hat{\lambda}$ is indistinguishable from λ to Π, then $\hat{\lambda}$ is a run. This run has the same failure pattern as λ, i.e., it is failure-free. Moreover, the failure detector behave perfectly. As a consequence, $\hat{\lambda} \in \mathscr{R}_n$. □

In the above lemma s' left-move with s [96], written $s \lhd s'$. By extension, we may deduce that for some run $\lambda\lambda'$ and some set of processes P,

Corollary 5. *If none of the messages received in $S = (\lambda'|P)$ was sent in $(\lambda' \setminus S)$, then $\lambda S(\lambda' \setminus S)$ is a nice run*

Corollary 6. *If none of the messages sent in $S = (\lambda'|P)$ is received in λ', then $\lambda(\lambda' \setminus S)S$ is a nice run.*

When A and B are sequences of steps, $A \lhd B$ denotes that B left-moves with A, that is $\forall (s,s') \in A \times B. s \lhd s'$.

Lemma 7. $\forall \lambda \in \mathscr{R}_n. \forall \lambda' \sqsubseteq \lambda. \lambda' \in \mathscr{R}_n$

Proof. Choose some run λ and some prefix $\lambda' \sqsubseteq \lambda$. By construction λ' is a run. Moreover as the failure detector behave perfectly in λ, it also behaves perfectly in λ'. From which it follows that λ' is a run. □

A.3 Proofs of Theorems 1 and 2

This section contains the proofs of the theorems stated in §3.2 which we deferred for readability.

▷ Theorem 1. Generic SMR reduces to Leaderless SMR.

Proof. We build a Generic SMR protocol A atop a Leaderless SMR protocol B as follows. Each node running protocol B holds a local copy of a partially ordered log L. This log is initially empty. Operation $submit(c)$ in A is mapped to operation $B.submit(c)$. When command c gets executed in B, we apply the following update to L: $L \leftarrow L \bullet c$. Clearly, this construction maintains that L is a partially ordered graph over time. Furthermore, at the light of the definition of the operator $\bullet$, it is easy to see that any two conflicting commands gets ordered. In addition, this construction satisfies the three properties that define Generic SMR, as shown below.

(Non-triviality) In algorithm B, a command c appears in some process dependency graph only if it was submitted before. Hence, in algorithm A, c is in L only if it was submitted before.

(Stability) Recall that this property holds when, for any partially ordered log L, at any point in time t, $L_t \sqsubseteq L_{t+1}$. The $\bullet$ operator does not remove edges or nodes, thus L_t is a subgraph of L_{t+1}. Now assume, for the sake of contradiction, that L_t does not prefix L_{t+1}. There must exist c in $L_t.V$ such that $(\mathsf{d},\mathsf{c}) \in L_{t+1}.E$ and $(\mathsf{d},\mathsf{c}) \notin L_t.E$. If $(\mathsf{d},\mathsf{c}) \notin L_t$ then $\mathsf{d} \in L_t.V$, as d cannot be added between time t and $t+1$ by definition of $\bullet$. This leads to $(\mathsf{d},\mathsf{c}) \notin L_{t+1}$; a contradiction.

(Consistency) We prove that for any two processes p and p', the set $\{L_p, L_{p'}\}$ is compatible. To achieve this, we show that $L = (L_p \cup L_{p'})$ is a partially ordered log that suffixes both L_p and $L_{p'}$. To this end, let us consider two conflicting commands c and d in L.

$\triangleright$ Claim 8. Commands c and d cannot be in $L_p.V \oplus L_q.V$.

Proof. By contradiction, assume c and d belong to different logs (wlog. say respectively L_p and $L_{p'}$). Applying Invariant 1 to p leads to the fact that $d \notin \mathit{deps}^*(\mathsf{c})$. Symmetrically from process p', we have that $\mathsf{c} \notin \mathit{deps}^*(\mathsf{d})$. A contradiction to the Consistency property of Leaderless SMR. □

$\triangleright$ Claim 9. For any process $q \in \{p, p'\}$, if (c,d) is in $L.E$ and d is in $L_q.V$ then (c,d) is in $L_q.E$.

Proof. Since $(\mathsf{c},\mathsf{d}) \in L.E$, (c,d) belongs to (say) $L_p.E$. This leads to $\mathsf{c} \rightarrow \mathsf{d}$ at p By Invariants 1 and 2, $\mathsf{c} \in \mathit{deps}^*(\mathsf{d})$ at p. Now, if $\mathsf{d} \in L_q.V$, q executes d. It follows that d was stable at q. By the Stability property of Leaderless SMR, $\mathsf{c} \in \mathit{deps}^*(\mathsf{d})$ at q. As a consequence, $\mathsf{c} \rightarrow \mathsf{d}$ at q and (c,d) is in $L_q.E$. □

The end of the proof goes as follows.

1. $L_p \sqsubseteq L \wedge L_{p'} \sqsubseteq L$
   From Claim 9.
2. L is a partially ordered log.
   (a) $\forall \mathsf{c},\mathsf{d} \in L.V. \mathsf{c} \asymp \mathsf{d} \Rightarrow ((\mathsf{c},\mathsf{d}) \in L.E \vee (\mathsf{d},\mathsf{c}) \in L.E)$
   From Claim 8.

(b) L is a directed acyclic graph.
If L is cyclic, by Claim 9, either L_p or $L_{p'}$ is cyclic. Contradiction.

□

▷ Theorem 2. Algorithm 2 implements Leaderless SMR.

Proof. Assume a run λ of Algorithm 2.

(Validity) Let c be a decided command at some process p in λ. By definition, $\mathit{deps}(\mathsf{c}) \neq \bot$ at process p. This is only possible through the execution of line 5 or line 9. If line 5 gets executed, then p took step $\mathit{submit}(\mathsf{c})$ previously. Otherwise, line 9 is executed. To execute line 9 it is necessary to receive a message (c, D) at line 8. Such a message is sent at line 6 by some process q. Executing line 6 implies that line 5 was executed by q before, which boils down to the previously analyzed case.

(Consistency) Let a and b be two conflicting committed commands at some process p. By definition, the two commands are committed when $\mathit{deps}(\mathsf{a}), \mathit{deps}(\mathsf{b}) \in 2^{\mathscr{C}}$ holds at p. Similarly to the Validity property, this is the case when line 5 or line 9 is executed. As seen in the proof of the validity property, executing line 9 can be traced back to the execution of line 5. Wlog. we only analyze the case where line 5 is executed hereafter.

By the Validity property of consensus, the value D assigned to $\mathit{deps}(\mathsf{a})$ at p was proposed before by a process q. The operation that yields D is executed at line 3 by q through the use of the DDS service. Similarly, we may define a process q' such that the value of $D' = \mathit{deps}(\mathsf{b})$ at p is the result of a call to $\mathit{announce}(\mathsf{b})$ by q'.

By the Visibility property of the DDS service, we have either $\mathsf{b} \in D$ or $\mathsf{a} \in D'$. Hence, the Consistency property of Leaderless SMR holds.

(Stability) Assume that c is eventually stable at processes p and p'. Let E and E' be the value of $\mathit{deps}(\mathsf{c})$ at respectively p and p' when this happens. In what follows, we prove that $E' = E$.

To establish this result, we first show that processes agree over aborted commands. Let q and q' be two processes that decide some command a. As with prior properties, we assume wlog. that such a decision is taken at line 5. Assume q aborts a. The output of line 5 depends on the computation at lines 3 and 4. Since command a is aborted, necessarily q takes the slow path. By the Validity property of CONS_a, $\top$ is proposed to CONS_a by some

process q''. This value is the response of *announce*(a) at line 2 by q''. Then, consider the following two cases. *(i)* If q' takes also the slow path, by the Agreement property of CONS_a, it should also abort a. *(ii)* Otherwise q' follows the fast path. In that case, by the Validity property of CONS_a, q' returns some value $(\_, \mathit{true})$ from *announce*(a) at line 2. Since q'' returns $\top$ from *announce*(a), this case contradicts the Weak Agreement property of the DDS service.

Now, let F and F' be the value of *deps*(c) first assigned by respectively p and p'. Wlog. assume that this assignment occurs at line 5. We show that for every command $\mathsf{a} \in F \oplus F'$, a, calling *announce*(a) returns $\top$. If p and p' take the slow path, then $F = F'$ by the Agreement property of CONS_c. Otherwise, one of the two processes, say p, takes the fast path. This implies that p returns (F, true) from *announce*(a). If p' takes also the fast path, (F', true) is returned from *announce*(a). Otherwise, by the Validity property of CONS_a, some process p'' returns (F', false) from *announce*(a). In both cases, the claim follows from the Weak Agreement property of the DDS service.

We now prove that F and F' converge toward E, the value of *deps*(c) when c is stable at p. These sets are bounded and the only update operation is the removal of an aborted command. Pick $\mathsf{a} \in F \oplus F'$. (Case $\mathsf{a} \in F \setminus F'$) Since c is stable, eventually a is decided. Thus, eventually p removes a from F. (Otherwise) Symmetrical to the previous one.

□

Appendix B

The ROLL Theorem

Below, we define formally the three properties introduced in §3.2.4. Then, we present two technical lemmas that characterize the behavior of Leaderless SMR protocols during nice runs. These lemmas are used to show the ROLL theorem and the chaining effect in the next section.

B.1 The Properties

In §3.2.4, we introduce Reliability, Optimal Latency and Load Balancing as three core properties of Leaderless SMR. Reliability guarantees that the protocol makes progress even if up to f failures occur. This means that once a command is submitted, the protocol must take a decision, possibly aborting it. Given a run λ, let us write $\mathsf{c} \in \lambda$ when *submit*(c) is invoked in λ. Then, we define this property as follows.

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

$$\forall \lambda \in \overline{\mathscr{R}}.\, \forall \mathsf{c} \in \lambda.\, \forall q \in \mathit{correct}(\lambda).\, \mathsf{faults}(\lambda) \leq f \Rightarrow \mathit{decide}_q(c) \in \lambda$$

Optimal Latency requires that in a nice run every command commits after two message delays. Moreover, in the absence of contention the command is immediately stable.

Message delays measure the time complexity of a sequence of steps, neglecting the cost of local computations. In detail, the latency of a causal path ρ, written $\Delta(\rho)$, is the number of consecutive *send*(m) then *recv*(m) steps in

ρ. Denoting $\prec$ the happens-before relation [87] in a run λ, $cpaths(\lambda)$ contains all the maximal chains in $(\lambda, \prec)$. The latency of λ is then defined as $\Delta(\lambda) = max\{\Delta(\rho) : \rho \in cpaths(\lambda)\}$.

To track contention during a run, we introduce function $contended(\lambda, \mathsf{c})$. This function returns *true* when there exists a command d conflicting with c such that d is submitted before c, and d is not committed at $coord(\mathsf{c})$ when c is submitted. Optimal Latency is then specified as follows.

(*Optimal Latency*) During a nice run, every call to $announce(d)$ returns a tuple (D, b) after two message delays such that *(i)* if there is no concurrent conflicting command to c, then b is set to *true*, *(ii)* $D \in 2^{\mathscr{C}}$, and *(iii)* for every $d \in D$, d was announced before.

$$\forall \lambda \in \mathscr{R}_n.\, \forall \mathsf{c} \in \mathscr{C}.\, resp(announce(\mathsf{c}), (D, b)) \in \lambda \Rightarrow \begin{array}{l} \wedge\ \Delta(\lambda|announce(\mathsf{c})) = 2 \\ \wedge\ (b = true \vee contended(\lambda, \mathsf{c})) \\ \wedge\ D \in 2^{\mathscr{C}} \\ \wedge\ \forall d \in D.\, \mathsf{inv}_p(announce(\mathsf{d})) \\ \qquad\qquad \prec_\lambda \mathsf{res}_q(announce(\mathsf{c})) \end{array}$$

An important property of Leaderless SMR protocols is to distribute the task of ordering conflicting commands across processes. This characteristic is captured by the Load Balancing property. In detail, this property requires that during a nice run *(i)* progress can be made using any fast path quorum, and *(ii)* when returning from the announcement of a command, no message gets undelivered.

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

$$\forall \lambda \in \mathscr{R}_n.\, \forall \mathsf{c} \notin \lambda.\, \forall Q \in FQuorums(\mathsf{c}).\, \exists \lambda'.\ \begin{array}{l} \wedge\ \lambda(\lambda'|announce(\mathsf{c})) \in \mathscr{R}_n \\ \wedge\ proc(\lambda'|announce(\mathsf{c})) = Q \\ \wedge\ pending(\lambda'|announce(\mathsf{c})) = \varnothing \end{array}$$

B.2 Characterizing ROLL protocols

Assume a ROLL protocol $\mathscr{P}$ and consider some nice run λ of $\mathscr{P}$ during which command c is announced. Since $\mathscr{P}$ ensures Optimal Latency, announcing c takes two message delays. This means that during the announcement of c a set of requests is sent by the coordinator to which a set of processes replies. The lemma below characterizes precisely this pattern, where Γ is a shorthand for $\lambda|announce(\mathsf{c})$.

Lemma 10. $\forall \mathscr{C} \in \mathit{cpaths}(\Gamma).\, \exists m, m', \rho, \rho', \rho''.\, \mathscr{C} = \mathsf{inv}_{\mathit{coord}(\mathsf{c})}(\mathit{announce}(\mathsf{c})) \cdot \rho \cdot \mathit{send}(m) \cdot \mathit{recv}(m) \cdot \rho' \cdot \mathit{send}(m') \cdot \mathit{recv}(m') \cdot \rho'' \cdot \mathsf{res}_{\mathit{coord}(\mathsf{c})}(\mathit{announce}(\mathsf{c}))$

Proof. Let s be the smallest element in $\mathscr{C}$. Since $\mathscr{C} \in \mathit{cpaths}(\Gamma)$ holds, we have that $s \in \lambda| \succeq \mathsf{inv}_{\mathit{coord}(\mathsf{c})}(\mathit{announce}(\mathsf{c}))$. Therefore $\mathsf{inv}_{\mathit{coord}(\mathsf{c})}(\mathit{announce}(c)) \preceq s$. The causal path $\mathscr{C}$ is a maximal chain in Γ, thus $s = \mathsf{inv}_{\mathit{coord}(\mathsf{c})}(\mathit{announce}(\mathsf{c}))$. Analogously, if we define s' as the largest element in $\mathscr{C}$, we have that $s' = \mathsf{res}_{\mathit{coord}(\mathsf{c})}(\mathit{announce}(\mathsf{c}))$.

Then, $\Delta(\mathscr{C}) = 2$ by Optimal Latency. As a consequence, there exists two messages m and m' and three sequences of steps ρ, ρ' and ρ'' such that *(i)* m is send by $\mathit{coord}(\mathsf{c})$ after $\mathsf{inv}_{\mathit{coord}(\mathsf{c})}(\mathit{announce}(\mathsf{c})) \cdot \rho$; *(ii)* m is received by some process q that executes the steps ρ' then sends a message m' to $\mathit{coord}(\mathsf{c})$; and *(iii)* $\mathit{coord}(\mathsf{c})$ receives m' and executes the steps ρ'' before the step $\mathsf{res}_{\mathit{coord}(\mathsf{c})}(\mathit{announce}(\mathsf{c}))$. □

At the light of the above characterization, we call *request* the first message exchanged in some causal path of Γ. Similarly, a reply is the answer received by $\mathit{coord}(\mathsf{c})$. We note m_c and m'_c respectively the last request sent and the first reply processed by the coordinator. The steps S_c, M_c and R_c below define a partitioning of Γ.

- S_c are the steps taken from announcing c to the sending of m_c at the coordinator. Formally, $S_\mathsf{c} = (\Gamma|\mathit{coord}(c)| \leq \mathit{send}(m_\mathsf{c}))$.
- R_c are the steps taken by $\mathit{coord}(c)$ after receiving m'_c until the announcement returns. In other words, $R_\mathsf{c} = (\Gamma|\mathit{coord}(c)| \geq \mathit{recv}(n_\mathsf{c}))$.
- M_c are the steps taken during the announcement of c which are neither in S_c, nor in R_c. That is, $M_\mathsf{c} = (\Gamma \setminus (S_\mathsf{c} \cup R_\mathsf{c}))$.

Based upon this partitioning, the two lemmas that follow characterize the behavior of ROLL protocols during nice runs. They are the basic building blocks of our two complexity results.

Lemma 11. $\forall \lambda \in \mathscr{R}_n.\, \forall \mathsf{c} \notin \lambda.\, \lambda S_\mathsf{c} M_\mathsf{c} R_\mathsf{c} \in \mathscr{R}_n \wedge \Delta(M_\mathsf{c}) = 0.$

Proof. Assume a command $\mathsf{c} \notin \lambda$ and some fast path quorum $Q \in \mathit{FQuorums}(\mathsf{c})$. By Load Balancing, we may suffix λ with λ' such that $\mathit{announce}(\mathsf{c})$ occurs in λ', only the processes in Q execute steps in $\lambda'|\mathit{announce}(\mathsf{c})$, and $\lambda(\lambda'|\mathit{announce}(c))$ is a nice run. From the definitions of S_c, M_c and R_c, λ' and $S_\mathsf{c} M_\mathsf{c} R_\mathsf{c}$ contain the same steps. It remains to show that taking such steps in this order remains legal.

Let us note $\Gamma = \lambda'|announce(c)$. Observe that none of the messages received in S_c was sent in $\Gamma \setminus S_c$. Otherwise, by Optimal Latency, c takes (at least) three message delays. Applying Corollary 5, $\lambda S_c(\Gamma \setminus S_c) \in \mathscr{R}_n$. In the same vein, none of the messages sent in R_c is received in $M_c = (\Gamma \setminus S_c) \setminus R_c$. Therefore we have that $\lambda S_c((\Gamma \setminus S_c) \setminus R_c)R_c \in \mathscr{R}_n$ by Corollary 6. By definition, $M_c = (\Gamma \setminus S_c) \setminus R_c$.

Now, assume by contradiction that $\Delta(M_c) > 0$. Let m be the message exchanged during M_c. Since $send(m)$ and $recv(m)$ belongs to Γ, there exists a causal path (C) starting with $\mathsf{inv}_{coord(c)}(announce(c))$ and ending with $\mathsf{res}_{coord(c)}(announce(c),)$ that contains these two steps. Applying Lemma 10, $\mathscr{C}$ is of the form $\mathsf{inv}_{coord(c)}(announce(c))\cdot \rho \cdot send(m) \cdot recv(m) \cdot \rho' \cdot send(m') \cdot recv(m') \cdot \rho'' \cdot \mathsf{res}_{coord(c)}(announce(c))$. Message m is neither a request, nor a reply, this implies that $\Delta(\rho') = 1$, and thus $\Delta(\mathscr{C}) \geq 3$, contradicting Optimal Latency. □

Lemma 12. $\forall \lambda \in \mathscr{R}_n. \forall P \subseteq \Pi. \lambda S_c(M_c|P) \in \mathscr{R}_n$

Proof. We start by picking $c \in \mathscr{C}$ such that $c \notin \lambda$. This allows us to apply Lemma 11 to achieve the following: $\lambda S_c M_c R_c \in \mathscr{R}_n$. Applying Lemma 7, we have that: $\lambda S_c M_c \in \mathscr{R}_n$. Then, let $X = M_c|P$, we know then that none of the messages received in X were sent in $M_c \setminus X$ (by Lemma 11, $\Delta(M_c) = 0$). Therefore, we can apply Corollary 5 to obtain $\lambda S_c X(M_c \setminus X) \in \mathscr{R}_n$. Finally, applying Lemma 7 gives us: $\lambda S_c(M_c|P) \in \mathscr{R}_n$. □

B.3 Proof

We now proceed to proving the ROLL theorem with the above two technical lemmas. The proof follows the sketch depicted in §3.3.

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

Proof. By contradiction, assume a ROLL protocol that satisfies $2F + f - 1 \leq n$. Let c_1 and c_2 be two conflicting commands in $\mathscr{C}$.

Define a partitioning P_1, P_2 and Q of Π, the set of processes, such that *(i)* $P_1 \cap P_2 = \emptyset$; *(ii)* $Q = \Pi(\setminus P_1 \cup P_2)$; *(iii)* $|P_1| = |P_2| = F - 1$; and *(iv)* $|Q| = n - 2(F-1)$.

The CAP impossibility result [71] tells us that $2F < n$. As a consequence, there exist at least two distinct processes p_1 and p_2 in Q. Let $Q_1 = P_1 \cap Q \setminus \{p_2\}$ and $Q_2 = P_2 \cup Q \setminus \{p_1\}$.

By applying Lemma 11 to the initial state, $\lambda_1 = S_1 M_1 R_1 \in \mathscr{R}_n$, By Optimal Latency, the fast path is taken in λ_1 and $deps(c_1) = \emptyset$ holds at p_1 at the end of the run. Symmetrically, we define $\lambda_2 = S_2 M_2 R_2 \in \mathscr{R}_n$.

By applying Lemma 12 to λ_1, $\lambda' = S_1(M_1|P_1) \in \mathscr{R}_n$. In a symmetrical manner, $\lambda'' = S_2(M_2|P_2) \in \mathscr{R}_n$. Observe that $proc(\lambda') \cap proc(\lambda'') = \emptyset$. Thus, by Lemma 3, we obtain $S_1(M_1|P_1)S_2(M_2|P_2) \in \mathscr{R}_n$. From Corollary 5, $\sigma = S_1S_2(M_2|P_2)(M_1|P_1) \in \mathscr{R}_n$.

Let $\hat{\sigma}$ be an infinite suffix of σ in which all the processes in Q are faulty. Such a suffix exist because we consider realistic failure detectors. As $|Q| \leq f$, c_1 and c_2 are eventually stable by Reliability. Let σ' be the shortest suffix of $\hat{\sigma}$ for which this is true at some process p. We define $\lambda_3 = \sigma\sigma'$. By construction, $proc(\sigma') \subseteq P_1 \cup P_2$.

In what follows, we construct a fourth run, λ_4. We will show that λ_4 is indistinguishable from λ_3 to p_2, while at the same time being indistinguishable from λ_1 to p_1. This implies that the same decision about $deps(\mathsf{c}_1)$ at p_1 in λ_1 is taken by p in λ_3.

By Lemma 11 applied to λ'', $S_2(M_2|P_2)S_1M_1R_1 \in \mathscr{R}_n$. Then, partitioning M_1 using Lemma 12, $S_2(M_2|P_2)S_1(M_1|P_1)(M_1|Q^* \cup \{p_1\})R_1 \in \mathscr{R}_n$. By applying Corollary 5, we obtain $\sigma(M_1|Q^* \cup \{p_1\})R_1 \in \mathscr{R}_n$.

Let λ_4 be a run with the same failure pattern and failure detector history as λ_3, and in which the steps $\sigma(M_1|Q^* \cup \{p_1\})R_1\sigma'$ are taken. Since $Q \cap (P_1 \cup P_2) = \emptyset$, λ_4 is not distinguishable from $\sigma(M_1|Q^* \cup \{p_1\})R_1$ to Q. Similarly, λ_4 is not distinguishable from $\sigma\sigma'$ to $P_1 \cup P_2$. Thus λ_4 is well-formed and a run of the protocol.

▷ Claim 13. $\lambda_1|\{p_1\} = \lambda_4|\{p_1\}$

Proof. $\lambda_1|p_1 = (S_1M_1R_1)|\{p_1\} = S_1(M_1|\{p_1\})R_1 = \lambda_4|\{p_1\}$ □

▷ Claim 14. $\lambda_4|p = \lambda_3|p$

Proof. $\lambda_4|p = (M_2|p)(M_1|p)(\sigma'|p) = \lambda_3|p$ □

Claim 13 implies that c_1 is stable at p_1 in λ_4 and that $deps(\mathsf{c}_1) = \emptyset$. Similarly, Claim 14 leads to c_1 stable at p in λ_4. In addition, the value of $deps(\mathsf{c}_1)$ at p in λ_4 is the same as in λ_3. By the Stability property of Leaderless SMR, $deps(\mathsf{c}_1) = \emptyset$ at p in λ_3.

A symmetric argument to the one above can be made using run λ_2 and a run λ_5. This leads to the conclusion that p decides $deps(\mathsf{c}_2) = \emptyset$ in λ_3.

From what precedes, run λ_3 contradicts the Consistency property of Leaderless SMR.

□

B.4 Chaining effect

This section shows how a chaining effect may occur in ROLL-optimal protocols. To establish this result, we construct a 2-asynchronous nice run with a pending chain of n commands. The run is built inductively starting from a solo run during which a single command is submitted.

At coarse grain, our construction works as follows. Let σ_i be a run with a chain of size i and some pending command $\mathsf{c_i}$. We extend σ_i with the partial announcement of a new command $\mathsf{c_{i+1}}$. Then, we take the decision for $\mathsf{c_i}$ to obtain σ_{i+1}. In σ_{i+1}, command $\mathsf{c_{i+1}}$ is pending, yet $\mathit{coord}(i)$ does not know if it is committed or not. We argue that in the latter case, it could have missed $\mathsf{c_i}$. Thus, $\mathit{coord}(i)$ must add $\mathsf{c_{i+1}}$ to the dependencies of $\mathsf{c_i}$, forming a new chain of size $i+1$.

B.4.1 An Inductive Reasoning

We build our result inductively using a family of k distinct commands $(\mathsf{c_i})_{i\in[1,k]}$. Each command is associated with a nice run (σ_i), a fast path quorum (Q_i), a subset of $f-1$ processes (P_i), and a process (q_i). We shall establish that at rank i the following property holds.

$$
\begin{aligned}
\mathfrak{P}(i) := \exists \sigma_i, Q_i, P_i, q_i, \Gamma_i, \Gamma_i'.\\
&\wedge\ \sigma_i = \Gamma_i S_i(M_i|P_i)\Gamma_i'(M_i|Q_i \setminus P_i)R_i\\
&\wedge\ \mathit{proc}(\Gamma_i') \cap Q_i = P_i\\
&\wedge\ (S_i(M_i|P_i)(\Gamma_i'|\Pi \setminus \{q_i\})) \lhd (\Gamma_i'|q_i)\\
&\wedge\ \sigma_i \vdash \mathsf{c_i c_{i-1}} \cdots \mathsf{c_1}\\
&\wedge\ \mathit{pending}(\sigma_i) = \varnothing\\
&\wedge\ \forall \rho \in \mathit{cpaths}(\Gamma_i').\, |\rho| \leq 1\\
&\wedge\ \sigma_i \in \text{2-asynchronous}
\end{aligned}
$$

In the above definition, the first two clauses give the general form of σ_i. The third clause indicates that the steps $(\Gamma_i'|q_i)$ left-move with $S_i(M_i|P_i)$. As we shall see later, these steps are taken by the coordinator of the next command (i.e., $\mathsf{c_{i+1}}$). The fourth clause requires that a chaining effect occurs with prior commands. The fifth and sixth clause upper-bound the asynchrony in σ_i. They are used to show by induction that the run is 2-asynchronous (last clause).

The remaining of this section is devoted to showing that $\mathfrak{P}(k)$ holds. From which we may deduce the following theorem (in §3.4).

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

Proof. The formal statement of the theorem is: $\forall k > 0. \exists \lambda \in$ 2-asynchronous. $\exists (\mathsf{c_i})_{i\in[1,n]} \subseteq \mathscr{C}. \lambda \vdash \mathsf{c_k c_{k-1}} \cdots \mathsf{c_1} \wedge \mathsf{c_n} \notin$ commit. To show this, let σ_k be the 2-asynchronous run given by $\Gamma_k S_k(M_k|P_k)\Gamma'_k(M_k|Q_k \setminus P_k)R_k$, by applying $\mathfrak{P}(k)$. Consider its prefix $\Gamma_k S_k(M_k|P_k)\Gamma'_k$. Command $\mathsf{c_k}$ is not committed in this run yet. Moreover, by $\mathfrak{P}(k)$, this run contains a chain $\mathsf{c_k c_{k-1}} \cdots \mathsf{c_1}$. □

B.4.2 Preliminaries

For $i = 1$, we chose Q_1 as any set of $(n - F)$ processes in Π. P_1 is any subset of $f - 1$ processes in Q_1. This set is constructable since by ROLL-optimality $n - F - (f - 1) = F - 2$ and by CAP $F > 1$. $coord(c_1)$ is any process in $Q_1 \setminus P_1$. Process q_1 is chosen outside of Q_1. Applying Lemma 12 then Lemma 11 leads to $S_1(M_1|P_1)(M_1|Q_1 \setminus P_1)R_1$. This gives us immediately $\mathfrak{P}(0)$ where both Γ_1 and Γ'_1 are empty.

Figure B.1 illustrates how we construct the quorum and process variables at rank $i+1$ from the ones at rank i. Such a construction ensures the following list of facts that are used to establish $\mathfrak{P}(i > 1)$.

(F1) $coord(\mathsf{c_{i+1}}) = q_i$

(F2) $P_{i+1} = Q_{i+1} \cap Q_i$

(F3) $P_{i+1} \cap P_i = \varnothing$

(F4) $q_{i+1} \notin Q_{i+1} \cup \{coord(\mathsf{c_i})\}$.

(F5) $(coord(\mathsf{c_{i-1}}) \cup P_{i+1}) \cap (coord(\mathsf{c_i}) \cup P_i) = \varnothing$

In detail, we build Q_{i+1} as $P_{i+1} \cup \hat{P}_{i+1} \cup \{coord(\mathsf{c_{i-1}})\}$, where: (a) $\hat{P}_{i+1}$ are $n - F - f + 1$ processes outside of Q_i; (b) P_{i+1} are $f - 1$ processes picked in $Q_i \setminus (P_i \cup coord(\mathsf{c_i}))$; and (c) $coord(\mathsf{c_{i+1}})$ is set to q_i. Then, process q_{i+1} is chosen in Q_i outside of $P_{i+1} \cup \{coordi\}$.

Let us establish the correctness of this construction. For starter, defining $\hat{P}_{i+1}$ is possible. Indeed, there are F processes outside of Q_i from which we need $n - F - f + 1$ of them. By ROLL-optimality, $F - (n - F - f + 1) = -n + 2F + f - 1 \geq 0$. Then, building P_{i+1} requires $f - 1$ processes in $Q_i \setminus (P_i \cup coord(\mathsf{c_i}))$. We have $n - F = -1 + f + F$ by ROLL and by CAP $F > 1$. Thus, there are

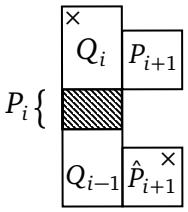


Figure B.1. The symbol × represents coordinator placement

enough processes. The above reasoning also tells us that we may pick q_{i+1} outside of $P_{i+1} \cup \{coordi\}$ in in Q_i. Then, (a) (F1) is true by construction; (b) Q_{i+1} is defined as $P_{i+1} \cup \hat{P}_{i+1} \cup \{coord(\mathsf{c}_{i-1})\}$. $\hat{P}_{i+1}$ does not intersect with Q_i. At rank i, $q_i \notin Q_i$ and $coord(\mathsf{c}_{i+1}) = q_i$. Thus, $coord(\mathsf{c}_{i+1}) \notin Q_i$. This establishes (F2); (c) By construction, we have (F3); and (d) q_{i+1} is chosen in Q_i outside of $P_{i+1} \cup \{coordi\}$. Hence, it is not in $\{Q_{i+1} \cup \{coord(\mathsf{c}_i)\}$, giving us (F4). (e) (F5) follows from the conjunction of (F3) and (F3).

B.4.3 Inductive step: $\mathfrak{P}(i) \Rightarrow \mathfrak{P}(i+1)$

Construction

Let us consider the prefix $\Gamma_i S_i(M_i|P_i)\Gamma_i'$ of σ_i. Applying Lemma 12, we obtain the following nice run: $\Gamma_i S_i(M_i|P_i)\Gamma_i' S_{i+1}(M_{i+1}|P_{i+1})$. As a consequence, from $\Gamma_i S_i(M_i|P_i)\Gamma_i'$ one may continue into either $(M_i|Q_i\ P_i)R_i$ or $S_{i+1}(M_{i+1}|P_{i+1})$. By (F5) we know that the sets $proc((M_i|Q_i \setminus P_i)R_i)$ and $proc(S_{i+1}(M_{i+1}|P_{i+1}))$ do not intersect. Applying Lemma 3, we obtain the run: $\Gamma_i S_i(M_i|P_i)\Gamma_i' S_{i+1}(M_{i+1}|P_{i+1})(M_i|Q_i \setminus P_i)R_i$.

Next, observe that from $\Gamma_i S_i(M_i|P_i)\Gamma_i' S_{i+1}(M_{i+1}|P_{i+1})$ both $(M_i|Q_i \setminus P_i)R_i$ and $(M_{i+1}|Q_{i+1} \setminus P_{i+1})R_{i+1}$ are possible. From (F1) and (F5), the sets $(Q_i \setminus P_i) \cup \{coord(\mathsf{c}_i)\}$ and $(Q_{i+1} \setminus P_{i+1}) \cup \{coord(\mathsf{c}_{i+1})\}$ are disjoint. By Lemma 3, $\sigma_{i+1} = \Gamma_{i+1} S_{i+1}(M_{i+1}|P_{i+1})\Gamma_{i+1}'(M_{i+1}|Q_{i+1} \setminus P_{i+1})R_{i+1}$ is a nice run , where $\Gamma_{i+1} = \Gamma_i S_i(M_i|P_i)\Gamma_i'$, and $\Gamma_{i+1}' = (M_i|Q_i \setminus P_i)R_i$.

Correctness

The claims that follow establish the correctness of the above construction with respect to the properties at rank $i+1$.

▷ Claim 15. $proc(\Gamma_{i+1}') \cap Q_{i+1} = P_{i+1}$

Proof. By construction, $proc(\Gamma_{i+1}') = (Q_i \setminus P_i) \cup coord(\mathsf{c}_i)$. On the other hand by (F2), $P_{i+1} = Q_i \cap Q_{i+1}$. The result follows from (F5). □

▷ Claim 16. $(S_{i+1}(M_{i+1}|P_{i+1})(\Gamma'_{i+1}|\Pi \setminus \{q_{i+1}\})) \lhd (\Gamma'_{i+1}|q_{i+1})$

Proof. By construction, $\Gamma'_{i+1} = (M_i|Q_i \setminus P_i)R_i$. Applying (F4), $q_{i+1} \notin Q_{i+1} \cup \{coord(\mathsf{c_i})\}$. Hence, $(\Gamma'_{i+1}|q_{i+1}) = (M_i|q_{i+1})$. Moreover, these steps left-move with $(\Gamma'_{i+1}|\Pi \setminus \{q_{i+1}\})$ by Lemma 11 and with $(S_{i+1}(M_{i+1}|P_{i+1})$ since $q_{i+1} \notin Q_{i+1}$. □

▷ Claim 17. $\sigma_{i+1} \vdash \mathsf{c_{i+1}} \rightarrow \mathsf{c_i}$

Proof. We now prove that a chaining effect occurs between the two commands $\mathsf{c_i}$ and $\mathsf{c_{i+1}}$ in σ_{i+1}. To this end, we first construct a nice run σ in which $\mathsf{c_{i+1}}$ is committed with $\mathsf{c_i} \notin deps(\mathsf{c_{i+1}})$. Then, we establish that σ is indistinguishable from σ_{i+1} to the coordinator of $\mathsf{c_i}$. This implies that $coord(\mathsf{c_i})$ must add $\mathsf{c_{i+1}}$ to the dependencies of $\mathsf{c_i}$ in σ_{i+1}.

With more details, our reasoning is as follows. First, consider $\Gamma_i S_i(M_i|P_i)\Gamma'_i$ that prefixes σ_i. Since $coord(\mathsf{c_{i+1}}) = q_i$ and $S_i(M_i|P_i) \lhd \Gamma'_i|q_i$ by our induction hypothesis, this run is equivalent to $\Gamma_i(\Gamma'_i|coord(\mathsf{c_{i+1}}))S_i(M_i|P_i)(\Gamma'_i|\Pi \setminus coord(\mathsf{c_{i+1}}))$. From which the prefix $\Gamma_i(\Gamma'_i|coord(\mathsf{c_{i+1}}))$ is obtained. Applying Lemma 11 to both $\mathsf{c_i}$ and $\mathsf{c_{i+1}}$ leads to $\sigma' := \Gamma_i(\Gamma'_i|coord(\mathsf{c_{i+1}}))(SMR)_{i+1}(SMR)_i$. This run clearly satisfies that $\mathsf{c_{i+1}} \rightarrow \mathsf{c_i}$.

The run σ is then derived from a series of rewriting of σ'. Applying Lemma 11 to $(SMR)_i$ then (F5) leads to $\Gamma_i(\Gamma'_i|coord(\mathsf{c_{i+1}}))S_i(M_i|P_i)(SMR)_{i+1}(M_i|Q_i \setminus P_i)R_i$. Following the same approach, $\Gamma_i(\Gamma'_i|coord(\mathsf{c_{i+1}}))S_i(M_i|P_i)S_{i+1}(M|P_{i+1})(M_i|Q_i \setminus P_i)R_i(M|Q_{i+1} \setminus P_{i+1})R_{i+1}$. The run σ is then defined as $\Gamma_i(\Gamma'_i|coord(\mathsf{c_{i+1}}))S_i(M_i|P_i)S_{i+1}(M|P_{i+1})(M_i|Q_i \setminus P_i)R_i$. We observe that, since $coord(\mathsf{c_i})$ takes the same steps in both σ and σ', then $\mathsf{c_{i+1}} \rightarrow \mathsf{c_i}$ holds in σ.

Let us then examine the steps of $coord(\mathsf{c_i})$ in the run σ_{i+1}. To this end, consider $\Gamma_i S_i(M_i|P_i)\Gamma'_i S_{i+1}(M|P_{i+1})(M_i|Q_i \setminus P_i)R_i$ that prefixes σ_{i+1}. By the assumption hypothesis $\mathfrak{P}(i)$, the steps $\Gamma'_i|coord(\mathsf{c_{i+1}})$ left-move with $S_i(M_i|P_i)(\Gamma'_i|\Pi \setminus coord(\mathsf{c_{i+1}}))$.

This leads to the run $\Gamma_i(\Gamma'_i|coord(\mathsf{c_{i+1}}))S_i(M_i|P_i)(\Gamma'_i|\Pi \setminus \{coord(\mathsf{c_{i+1}})\})S_{i+1}(M|P_{i+1})(M_i|Q_i \setminus P_i)R_i$.

Applying again the assumption hypothesis, $proc(\Gamma'_i|\Pi \setminus \{coord(\mathsf{c_{i+1}})\}) \cap Q_i = \varnothing$. This leads to the following equivalent run $\Gamma_i(\Gamma'_i|coord(\mathsf{c_{i+1}}))S_i(M_i|P_i)S_{i+1}(M|P_{i+1})(M_i|Q_i \setminus P_i)R_i(\Gamma'_i|\Pi \setminus \{coord(\mathsf{c_{i+1}})\})$. We pick the prefix $\Gamma_i(\Gamma'_i|coord(\mathsf{c_{i+1}}))S_i(M_i|P_i)S_{i+1}(M|P_{i+1})(M_i|Q_i \setminus P_i)R_i$. In this prefix, $coord(\mathsf{c_i})$ takes the same steps as in σ. Hence, $\mathsf{c_{i+1}} \rightarrow \mathsf{c_i}$ holds in σ_{i+1}. □

▷ Claim 18. $pending(\sigma_{i+1}) = \varnothing$

Proof. By induction, no message is pending in σ_i. Consider then a message sent in $(SMR)_{i+1}$. By Load-Balancing, this message is received in this sequence of steps. □

▷ Claim 19. $\forall \rho \in cpaths(\Gamma'_{i+1}).|\rho| \leq 1$

Proof. This claims follows from the definition of Γ'_{i+1}. □

▷ Claim 20. $\sigma_{i+1} \in$ 2-asynchronous

Proof. Assume that a message m is sent in σ_{i+1}. From Claim 19, m is received in σ_{i+1}. Below, we conduct a case analysis depending on the position of $recv(m)$ in σ_{i+1}. Our analysis shows that any path ρ concurrent to m is at most of length two.

- (Γ_{i+1}) If ρ is concurrent to m in σ_{i+1}, it is also concurrent to m in σ_i. The induction hypothesis implies that $|\rho| \leq 2$.
- ($S_{i+1}(M_{i+1}|P_{i+1})$ σ_i does not contain a pending message. Thus, m is sent by $coordi+1$ in S_{i+1}. Applying Lemma 11, $\Delta(M_{i+1}) = 0$ implies that $|\rho| = 1$.
- (Γ'_{i+1}) By Load-Balancing, m is sent in $(SMR)_i$. If ρ ends in Γ'_i, then by the induction hypothesis ρ is at most of size 2. Otherwise, the last message in ρ is sent in $S_{i+1}(M_{i+1}|P_{i+1})$ by the coordinator of c_{i+1}. In that case, $(\Gamma'_i|coordi+1) \lhd S_i(M_i|P_i)(\Gamma'_i|\Pi \setminus coordi+1)$ implies that $|\rho| = 1$.
- $((M_{i+1}|Q_{i+1} \setminus P_{i+1})R_{i+1})$ $send(m)$ occurs during the same sequence of steps, or it happens in $S_{i+1}(M_{i+1}|P_{i+1})$. The former case leads to $|\rho| = 1$. In the later, as no message sent in Γ'_{i+1} is pending, ρ is fully included in $S_{i+1}(M_{i+1}|P_{i+1})\Gamma_{i+1}$. By Claim 19, every such path in is (at most) of size 2.

□

Appendix C

Linearizable objects with BLSMR

In what follows, we first introduce some preliminary notions. Then, we explain how to implement any linearizable data type atop BLSMR. Our construction follows closely the one in [63], with a twist to accomodate byzantine processes.

C.1 Preliminaries

We base our reasoning and algorithms upon the notion of trace [53]. Two words in a class contain the same commands and sort non-commuting ones in the same order. A trace can be seen as as special case of the notion of c-struct used to define the generalized consensus problem [90].

State machine.. We assume a sequential object specified by the following components: *(i)* a set of states $\mathscr{S}$; *(ii)* an initial state $s_0 \in \mathscr{S}$; *(iii)* a set of commands $\mathscr{C}$ that can be performed on the object; *(iv)* a set of their response values $\mathscr{V}$; and *(v)* a transition function $\tau : \mathscr{S} \times \mathscr{C} \to \mathscr{S} \times \mathscr{V}$. In the following, we use special symbols $\bot$ and $\top$ that do not belong to $\mathscr{V}$. When applying a command, we use .st and .val selectors to respectively extract the state and the response value, i.e., given a state s and a command c, we let $\tau(s,c) = (\tau(s,c).\mathsf{st}, \tau(s,c).\mathsf{val})$. Without lack of generality, we consider that commands are applicable to every state. A command c is a *read* if it does not change the object state: $\forall s.\, \tau(s,c).\mathsf{st} = s$; otherwise, c is a *write*. We denote by Read and Write the set of read and write commands.

Command words.. A *command word* x is a sequence of commands. The empty word is denoted 1 and $\mathscr{C}^*$ is the set of all command words. We use the following notations for a word x: $|x|$ is the length of x; $x[i \geq 1]$ is the i-th element in x; $|x|_c$ is the number of occurrences of command c in x. We write $c^i \in x$ when c occurs at least $i > 0$ times in x. $pos(c^i, x)$ is the position of

the i-th occurrence of command c in x, with $pos(c^i, x) = 0$ when $c^i \notin x$. The shorthand $c^i <_x d^j$ stands for $pos(c^i, x) < pos(d^j, x)$. The set $\mathsf{cmd}(x)$ is defined as $\{(c, i) : c^i \in x\}$. The operator $x \setminus c$ deletes the last occurrence of c in x (if such an occurrence exists). By extension, for some word y, $x \setminus y$ applies $x \setminus c$ for every $(c, i) \in \mathsf{cmd}(y)$. We let $\sqsubseteq$ be the prefix relation induced by the append operator over $\mathscr{C}^*$. The prefix of x up to some occurrence c^i is the command word $x|_{\leq c^i}$. If $c^i \notin x$, then by convention $x|_{\leq c^i}$ equals 1. In case c appears once in x, $x|_{\leq c}$ is a shorthand for $x|_{\leq c^1}$.

Lemma 21. *Consider a command c and two words x and y. Then, $|xy|_c$ equals $|x|_c + |y|_c$. Moreover, if $c^k \in xy$ then $pos(c^k, xy)$ equals $pos(c^k, x)$, if $k \leq |x|_c$ and $|x| + pos(c^{k-|x|_c}, y)$ otherwise.*

Proof. Follows from the definitions. □

Equivalence of command words.. We define function τ^* by the repeated application of τ. In detail, for a state s we define $\tau^*(s, 1) = (s, nil)$, for some symbol $nil \in \mathscr{V}$, and if x is non-empty then we have:

$$\tau^*(s, x) = \begin{cases} \tau(s, x[1]), & \text{if } |x| = 1; \\ \tau^*(\tau(s, x[1]).\mathsf{st}, x[2] \dots x[n]), & \text{otherwise.} \end{cases}$$

Two commands c and d *commute*, written $c \not\asymp d$, if in every state s we have:

$$\begin{array}{l} \tau^*(s, cd).\mathsf{st} = \tau^*(s, dc).\mathsf{st}; \\ \tau^*(s, dc).\mathsf{val} = \tau^*(s, c).\mathsf{val}; \\ \tau^*(s, cd).\mathsf{val} = \tau^*(s, d).\mathsf{val}. \end{array}$$

Relation $\not\asymp$ is an equivalence relation over $\mathscr{C}$. We write $c \not\parallel d$ the fact that c and d do no commute. Two words $x, y \in \mathscr{C}^*$ are *equivalent*, written $x \sim y$, when there exist words $z_1, \dots, z_{k \geq 1}$ such that $z_1 = x$, $z_k = y$ and for all i, $1 \leq i < k$, there exist words z', z'' and commands $c \not\asymp d$ satisfying $z_i = z'cdz''$, $z_{i+1} = z'dcz''$. This means that a word can be obtained from another by successive transpositions of neighboring commuting commands. One may show that $u \sim v$ holds when u and v contain the same commands and order non-commuting ones the same way. In such a case, commands have the same effects.

Lemma 22 ([53])**.** *Relation $x \sim y$ holds iff $\mathsf{cmd}(x) = \mathsf{cmd}(y)$ and for any $c \not\parallel d$, $c^i <_x d^j \Leftrightarrow c^i <_y d^j$.*

Lemma 23. *If $x \sim y$ then for every command c, $\tau^*(s_0, x|_{\leq c^i}).\mathsf{val} = \tau^*(s_0, y|_{\leq c^i}).\mathsf{val}$.*

Proof. We show that the proposition holds if $x = z'abz''$ and $y = z'baz''$, for $a \not\asymp b$ and words z' and z''. Obviously, this is true for any command c in z'. Now, if $a = c^i$, then the proposition holds by definition of relation $\not\asymp$. A symmetric argument holds for $b = c^i$. Then, because a and b are commuting, we may observe that $\tau^*(s_0, z'ab).\mathsf{st} = \tau^*(s_0, z'ba).\mathsf{st}$. From which, we deduce that the result also holds if $c^i \in z''$. Now, applying the above claim to the definition of $x \sim y$, we deduce that the proposition holds in the general case. □

Command traces.. The equivalence class of x for the relation $\sim$ is denoted $[x]$. This is the set of words that order non-commuting commands in the same way as x. Hereafter, we note Traces the quotient set of $\mathscr{C}^*$ by relation $\sim$. An element in Traces is named a *command trace*. For any $x, y, z \in \mathscr{C}^*$, it is easy to observe that if $x \sim y$ holds, then both $(zx \sim zy)$ and $(xz \sim yz)$ are true. As a consequence, $\sim$ is a congruence relation over $\mathscr{C}^*$. It follows that Traces together with the append operator defined as $[x][y] = [xy]$ forms a monoid[1]. Now, consider the natural ordering induced by the append operator on Traces. In other words, $[x] \sqsubseteq [y]$ holds iff $[x][z] = [y]$ for some $[z]$. One can show that relation $\sqsubseteq$ is a partial order over Traces [53].

Lemma 24. *If* $[x] \sqsubseteq [y]$*, then* $[x][y \setminus x] = [y]$.

Proof. From $[x] \sqsubseteq [y]$, there exists some z such that $[x][z] = [y]$. We show that $[y \setminus x] = [z]$. If $c^i \in y$ and $c^i \notin x$, by Lemma 22, $c^i \in z$. Conversely, if $c^i \in z$ then $c^i \notin x$ and by Lemma 22, $c^i \in y$. Then, by applying again Lemma 22, we deduce that $c^i <_z d^j \Leftrightarrow c^i <_{y \setminus x} d^j$. □

Lemma 25. *If* $\mathsf{cmd}(x) \subseteq \mathsf{cmd}(y)$ *and for any* $c \not\asymp d$*,* $c^i <_y d^j \wedge d^j \in x \Rightarrow c^i <_x d^j$*, then* $[x] \sqsubseteq [y]$.

Proof. By Lemma 21, $\mathsf{cmd}(x(y \setminus x)) = \mathsf{cmd}(y)$. Then, choose $c, d \in \mathscr{C}$ with $c \not\asymp d$ and $c^i <_y d^j$. We show that $c^i <_{x(y \setminus x)} d^j$. Let $k = |x|_c$ and $l = |x|_d$. (Case $l = j$) By assumption. (Otherwise) If $k = i$ then $c^i \in x$ and $d^{j-l} \in (y \setminus x)$. In the converse case, c^{i-k} and d^{j-l} are both in $(y \setminus x)$. We then conclude by applying Lemma 21. □

Lemma 26. *If* $[x] \sqsubseteq [y]$*, then for every command* c *with* $c^i \in x$*,* $\tau^*(s_0, x|_{\leq c^i}).\mathsf{val} = \tau^*(s_0, y|_{\leq c^i}).\mathsf{val}$.

Proof. From Lemma 24, $x(y \setminus x) \sim y$. Choose $c^i \in x$. By Lemma 23, $\tau^*(s_0, x(y \setminus x)|_{\leq c^i}).\mathsf{val} = \tau^*(s_0, y|_{\leq c^i}).\mathsf{val}$. Since $c^i \in x$, $c^i \notin (y \setminus x)$ and $x(y \setminus x)|_{\leq c^i} = x|_{\leq c^i}$. □

[1] Gerard Lallement. *Semigroups and Combinatorial Applications*. John Wiley & Sons, Inc., 1979.

Histories.. A *history* is a sequence of *events* of the form $\mathsf{inv}_i(c)$ or $\mathsf{res}_i(c, v)$, where $i \in \Pi$, $c \in \mathscr{C}$ and $v \in \mathscr{V}$. The two kinds of events denote respectively an *invocation* of command c by process i, and a *response* to this command returning some value v. We write $c \prec_h d$ the fact that the response of c precedes the invocation of command d in history h. For a process i, we let $h|i$ be the projection of history h onto the events by i. The following classes of histories are of particular interest:

- A history h is *sequential* if it is a non-interleaved sequence of invocations and matching responses, possibly terminated by a non-returning invocation.
- A history h is *well-formed* if *(i)* $h|i$ is sequential for every $i \in \Pi$; *(ii)* each command c is invoked at most once in h; and *(iii)* for every response $\mathsf{res}_i(c, v)$, an invocation $\mathsf{inv}_i(c)$ occurs before in h.
- A well-formed history h is *complete* if every invocation has a matching response. We shall write *complete*(h) the largest complete prefix of h.
- A well-formed history h is *legal* if h is complete and sequential and for any command c, if a response value appears in h, then it equals $\tau^*(s_0, h|_{\leq c}).\mathsf{val}|$.

Linearizability.. Two histories h and h' are *equivalent*, written $h \sim h'$, if they contain the same set of events. History h is *linearizable* [80] when it can be extended (by appending zero or more responses) into some history h' such that *complete*(h') is equivalent to a legal and sequential history l preserving the real-time order in h, i.e., $\prec_h \subseteq \prec_l$.

C.2 Algorithm

Algorithm 6 presents the pseudo-code of our construction atop BLSMR. Each line of this algorithm is atomic. To execute a command c on the shared object, a client process executes `invoke`(c). This sends a message to $f + 1$ replicas. Upon receiving such a message, a process submits command c to BLSMR. Once executed, the result of the command is then sent back to the client. The client considers such a value once it has been received from $f + 1$ replicas.

Algorithm 6 employs the following variables:

- B is an instance of BLSMR abstraction together with the execution algorithm depicted in Algorithm 1. The $\asymp$ relation is set to an over-approximation of the non-commutativity relation among commands ($\not\parallel$).
- S is a local copy of the state of the sequential object under concern. Initially, it equals s_0.
- Variable λ stores the log of commands applied to the local copy.

Algorithm 6 Linearizable objects with BLSMR – code at process i.

```
1:  Local Variables:
2:      L                                  // An instance of BLSMR-E, with c ∦ d ⇒ c ≍ d.
3:      S ← s_0                            // A local copy of the sequential object.
4:      λ ← 1                              // A command word.
5:
6:  invoke(c) :=                           // Client
7:      eff: choose Q ⊆ Π such that |Q| = f + 1
8:           send(c) to Q
9:
10: respond(c) :=
11:     pre: ∃v ∈ V, Q ⊆ Π. |Q| = f + 1 ∧ ∀q ∈ Q. recv(c, v) from q
12:     eff: return v
13:
14: invoke(c) :=                           // Replica
15:     pre: recv(c) from client(c)
16:     eff: B.submit(c)
17:
18: respond(c) :=
19:     pre: B.execute(c)
20:     eff: λ ← λ • c; (S, v) ← τ(S, c)
21:          send(c, v) to client(c)
```

C.3 Correctness

In what follows, λ is a run of Algorithm 6 and h the corresponding history. For some variable *var*, we denote by var_i the value of *var* at process i. The notation var_i^λ refers to the value of var_i at the end of the execution λ. For starters, we prove that at any point in time a single occurrence of a command may appear in λ_i.

Proposition 27. $\forall i \in Correct.\ \Box(\forall c \in \mathscr{C}.\ |\lambda_i|_c \leq 1)$.

Proof. (by induction) λ_i is initially empty. Then, assume that a correct process i appends c to λ_i at line 20. Command c is thus executed by BLSMR at line 19. From the pseudo-code of Algorithm 1, this happens at most once. Hence, $(|\lambda_i|_c = 1)$ is true from that point in time. □

The execution mechanism at line 20 applies in order the commands of λ_i to update S_i. Such an approach maintains the following two invariants:

Proposition 28. $\forall i \in Correct.\ \Box(S_i = \tau^*(s_0, \lambda_i).\mathsf{st})$.

Proof. (by induction.) Initially $\lambda_i = 1$, leading to $\tau^*(s_0, \lambda_i).\mathsf{st} = s_0$. This coincides with the value of S_i at start time. At line 20, variable S_i is changed to ${S_i}' = \tau^*(S_i, \lambda_i').\mathsf{st}$, with $\lambda_i' = \lambda_i \bullet c$. By induction, $S_i = \tau^*(s_0, \lambda_i).\mathsf{st}$. It follows that:

$$\begin{aligned} {S_i}' &= \tau^*(S_i, \lambda_i \bullet c).\mathsf{st} \\ &= \tau^*(\tau^*(s_0, \lambda_i).\mathsf{st}, c).\mathsf{st} \\ &= \tau^*(s_0, \lambda_i').\mathsf{st} \end{aligned}$$

□

Proposition 29. $\forall \mathsf{res}_{client(c)}(c, v) \in h, \exists i \in Correct.\ v = \tau^*(s_0, \lambda_i^{\lambda}|_{\leq c}).\mathsf{val}$.

Proof. From line 11, $client(c)$ received $f + 1$ times the response value v. This implies that it received such a value from a correct process, say i. From the code at line 21, v is the result of the computation at lines 19 to 21. Let λ be the value of λ_i before this execution. Applying Proposition 28 leads to $S_i = \tau^*(s_0, \lambda).\mathsf{st}$. Thus, we have $v = \tau^*(s_0, \lambda \bullet c).\mathsf{val}$. By Proposition 27, $\lambda_i^{\lambda}|_{\leq c} = \lambda \bullet c$. Thus, the claim holds. □

The above proposition explains how the response values of h are computed. We now construct a linearization of the commands submitted to BLSMR that is consistent with these return values.

To this end, we first establish that commands are executed at the correct processes in a sound order, similarly to [63]. Consider that $c \mapsto_i d$ holds if some correct process i executes c before d. We define $\mapsto$ as the order in which the correct replicas execute commands in Algorithm 6, that is $\mapsto = \bigcup_{j \in Correct} \mapsto_j$.

Proposition 30. *The transitive closure of $\mapsto$ forms an order over $\mathscr{C}$.*

Proof. By Invariants 1 and 2 of BLSMR and Proposition 28. □

Then, we linearize the commands executed during history h as explained below. Hereafter, we shall note this linearization δ.

▷ CONSTRUCTION 1. Initially, δ is set to 1. Let E be the set $\bigcup_{j \in Correct} \mathsf{cmd}(\lambda_j^{\lambda})$. We append each command $c \in E$ to δ following some linear extension of the $\mapsto$ relation over E.

Proposition 31. $\forall i \in Correct.\ [\lambda_i^{\lambda}] \sqsubseteq [\delta]$.

Proof. For any λ_i^λ, we have $\mathsf{cmd}(\lambda_i^\lambda) \subseteq \mathsf{cmd}(\delta)$. Now consider a pair of non-commuting command (c, d) in δ, with $c <_\delta d$ and $d \in \lambda_i^\lambda$. Observe that if $c \notin \lambda_i^\lambda$ or $d <_{\lambda_i^\lambda} c$, then $d \mapsto_i c$ holds; thus, we have necessarily $c <_{\lambda_i^\lambda} d$. Applying Lemma 25, $[\lambda_i^\lambda] \sqsubseteq [\delta]$. □

Consider the complete, sequential and legal history l produced by applying the commands in δ to s_0 following the order $<_\delta$. For every pending command c in h, if c has no response v in h, we append $\mathsf{res}_i(c, v)$ to h, where i is the caller of c and v the response of c in l. Name h' the resulting history that by construction completes h.

Proposition 32. $l \sim h'$

Proof. By applying Proposition 29, Proposition 31 and Lemma 26. □

Proposition 33. $\prec_h \subseteq \prec_l$

Proof. By construction of l and the fact that $\prec_h \subseteq <_\lambda$. □

At the light of the last two propositions, we may conclude the result that follows.

Theorem 34. *For every run λ of Algorithm 6, the history h induced by λ is linearizable.*

Bibliography

[1] Abbadi, A. E. and Toueg, S. [1989]. Maintaining availability in partitioned replicated databases, *ACM Trans. Database Syst.* **14**(2): 264–290.
URL: *https://doi.org/10.1145/63500.63501*

[2] Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M. K. and Wylie, J. J. [2005]. Fault-scalable byzantine fault-tolerant services, *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, Association for Computing Machinery, New York, NY, USA, pp. 59–74.
URL: *https://doi.org/10.1145/1095810.1095817*

[3] Abraham, I., Gueta, G. and Malkhi, D. [2018]. Hot-stuff the linear, optimal-resilience, one-message BFT devil, *CoRR* **abs/1803.05069**.
URL: *http://arxiv.org/abs/1803.05069*

[4] Abraham, I., Gueta, G., Malkhi, D., Alvisi, L., Kotla, R. and Martin, J. [2017]. Revisiting fast practical byzantine fault tolerance, *CoRR* **abs/1712.01367**.
URL: *http://arxiv.org/abs/1712.01367*

[5] Abraham, I., Malkhi, D. and Spiegelman, A. [2019]. Asymptotically optimal validated asynchronous byzantine agreement, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, Association for Computing Machinery, New York, NY, USA, pp. 337–346.
URL: *https://doi.org/10.1145/3293611.3331612*

[6] Abraham, I., Nayak, K., Ren, L. and Xiang, Z. [2021]. Brief note: Fast authenticated byzantine consensus, *CoRR* **abs/2102.07932**.
URL: *https://arxiv.org/abs/2102.07932*

[7] Abraham, I. and Stern, G. [2020]. Information theoretic hotstuff, *CoRR* **abs/2009.12828**.
URL: *https://arxiv.org/abs/2009.12828*

[8] Agrawal, D. and El Abbadi, A. [1991]. An efficient and fault-tolerant solution for distributed mutual exclusion, *ACM Trans. Comput. Syst.* **9**(1): 1–20.
URL: *https://doi.org/10.1145/103727.103728*

[9] Aguilera, M. K. and Strom, R. E. [2000]. Efficient atomic broadcast using deterministic merge, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, Association for Computing Machinery, New York, NY, USA, pp. 209–218.
URL: *https://doi.org/10.1145/343477.343620*

[10] Amir, Y., Danilov, C., Dolev, D., Kirsch, J., Lane, J., Nita-Rotaru, C., Olsen, J. and Zage, D. [2010]. Steward: Scaling byzantine fault-tolerant replication to wide area networks, *IEEE Transactions on Dependable and Secure Computing* **7**(1): 80–93.

[11] Anceaume, E., Del Pozzo, A., Ludinard, R., Potop-Butucaru, M. and Tucci-Piergiovanni, S. [2019]. Blockchain abstract data type, *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, Association for Computing Machinery, New York, NY, USA, pp. 349–358.
URL: *https://doi.org/10.1145/3323165.3323183*

[12] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S. W. and Yellick, J. [2018]. Hyperledger fabric: A distributed operating system for permissioned blockchains, *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/3190508.3190538*

[13] Anger, F. D. [1989]. On lamport's interprocessor communication model, *ACM Trans. Program. Lang. Syst.* **11**(3): 404–417.
URL: *https://doi.org/10.1145/65979.65982*

[14] Anta, A. F., Konwar, K., Georgiou, C. and Nicolaou, N. [2018]. Formalizing and implementing distributed ledger objects, *SIGACT News* **49**(2): 58–76.
URL: *https://doi.org/10.1145/3232679.3232691*

[15] Antoniadis, K., Guerraoui, R., Malkhi, D. and Seredinschi, D. [2018]. State machine replication is more expensive than consensus, *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA,*

October 15-19, 2018, pp. 7:1–7:18.
URL: *https://doi.org/10.4230/LIPIcs.DISC.2018.7*

[16] Arun, B., Peluso, S., Palmieri, R., Losa, G. and Ravindran, B. [2017]. Speeding up consensus by chasing fast decisions, *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 49–60.

[17] Aublin, P.-L., Guerraoui, R., Knežević, N., Quéma, V. and Vukolić, M. [2015]. The next 700 bft protocols, *ACM Trans. Comput. Syst.* **32**(4).
URL: *https://doi.org/10.1145/2658994*

[18] Barak, B., Canetti, R., Lindell, Y., Pass, R. and Rabin, T. [2007]. Secure computation without authentication. An extended abstract of this work appeared in Crypto 2005. canetti@csail.mit.edu 13864 received 16 Dec 2007.
URL: *http://eprint.iacr.org/2007/464*

[19] Bazzi, R. A. and Ding, Y. [2004]. Non-skipping timestamps for byzantine data storage systems, *in* R. Guerraoui (ed.), *Distributed Computing*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 405–419.

[20] Bazzi, R. A. and Herlihy, M. [2019]. Clairvoyant state machine replication, *CoRR* **abs/1905.11607**.
URL: *http://arxiv.org/abs/1905.11607*

[21] Ben-Or, M. [1983]. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols, *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, Association for Computing Machinery, New York, NY, USA, pp. 27–30.
URL: *https://doi.org/10.1145/800221.806707*

[22] Bentov, I., Lee, C., Mizrahi, A. and Rosenfeld, M. [2014]. Proof of activity: Extending bitcoin's proof of work via proof of stake [extended abstract], *SIGMETRICS Perform. Eval. Rev.* **42**(3): 34–37.
URL: *https://doi.org/10.1145/2695533.2695545*

[23] Bessani, A., Alchieri, E., Sousa, J., Oliveira, A. and Pedone, F. [2020]. From byzantine replication to blockchain: Consensus is only the beginning, *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*, IEEE, pp. 424–436.
URL: *https://doi.org/10.1109/DSN48063.2020.00057*

[24] Bessani, A., Sousa, J. and Alchieri, E. E. [2014]. State machine replication for the masses with bft-smart, *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 355–362.

[25] Bitansky, N. [2017]. Verifiable random functions from non-interactive witness-indistinguishable proofs, *in* Y. Kalai and L. Reyzin (eds), *Theory of Cryptography*, Springer International Publishing, Cham, pp. 567–594.

[26] *Bitcoin Confirmation Time* [2021]. https://en.bitcoin.it/wiki/Confirmation.

[27] *Bitcoin energy consumption* [2019]. https://digiconomist.net/bitcoin-energy-consumption.

[28] *Bitcoin energy consumption* [2021]. https://algorand.foundation/faq.

[29] *Bitcoin Full Nodes* [2021]. https://coin.dance/nodes.

[30] Bodlaender, M. H. L., Halldórsson, M. M., Konrad, C. and Kuhn, F. [2016]. Brief announcement: Local independent set approximation, *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pp. 93–95.
URL: *https://doi.org/10.1145/2933057.2933068*

[31] Börzsönyi, S., Kossmann, D. and Stocker, K. [2001]. The skyline operator, *Proceedings of the 17th International Conference on Data Engineering*, IEEE Computer Society, USA, pp. 421–430.

[32] Bracha, G. [1987]. Asynchronous byzantine agreement protocols, *Information and Computation* **75**(2): 130–143.
URL: *https://www.sciencedirect.com/science/article/pii/089054018790054X*

[33] Bravo, M., Chockler, G. and Gotsman, A. [2020]. Making Byzantine Consensus Live, *in* H. Attiya (ed.), *34th International Symposium on Distributed Computing (DISC 2020)*, Vol. 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 23:1–23:17.
URL: *https://drops.dagstuhl.de/opus/volltexte/2020/13101*

[34] Buchman, E., Kwon, J. and Milosevic, Z. [2018]. The latest gossip on BFT consensus, *CoRR* **abs/1807.04938**.
URL: *http://arxiv.org/abs/1807.04938*

[35] Burrows, M. [2006]. The chubby lock service for loosely-coupled distributed systems, *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, USENIX Association, Seattle, WA.
URL: *https://www.usenix.org/conference/osdi-06/chubby-lock-service-loosely-coupled-distributed-systems*

[36] Buterin, V. and Griffith, V. [2017]. Casper the friendly finality gadget, *CoRR* **abs/1710.09437**.
URL: *http://arxiv.org/abs/1710.09437*

[37] Cachin, C., Kursawe, K., Petzold, F. and Shoup, V. [2001]. Secure and efficient asynchronous broadcast protocols, *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, Springer-Verlag, Berlin, Heidelberg, pp. 524–541.

[38] Cachin, C., Kursawe, K. and Shoup, V. [2005]. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography, *J. Cryptol.* **18**(3): 219–246.
URL: *https://doi.org/10.1007/s00145-005-0318-0*

[39] Canakci, B. and Van Renesse, R. [2021]. Scaling membership of byzantine consensus, *ACM Trans. Comput. Syst.* **38**(3-4).
URL: *https://doi.org/10.1145/3473138*

[40] Cason, D., Milosevic, N., Milosevic, Z. and Pedone, F. [2021]. *Gossip Consensus*, Association for Computing Machinery, New York, NY, USA, p. 198–209.
URL: *https://doi.org/10.1145/3464298.3493395*

[41] Castro, M. and Liskov, B. [1999]. Practical byzantine fault tolerance, *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, USENIX Association, USA, pp. 173–186.

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

[43] Civit, P., Gilbert, S. and Gramoli, V. [2021]. Polygraph: Accountable byzantine agreement, *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pp. 403–413.

[44] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle,

D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R. and Woodford, D. [2012]. Spanner: Google's globally-distributed database, *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 251–264.

[45] Cowling, J., Myers, D., Liskov, B., Rodrigues, R. and Shrira, L. [2006]. Hq replication: A hybrid quorum protocol for byzantine fault tolerance, *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, USENIX Association, USA, pp. 177–190.

[46] Crain, T., Gramoli, V., Larrea, M. and Raynal, M. [2018]. Dbft: Efficient leaderless byzantine consensus and its application to blockchains, *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pp. 1–8.

[47] Crain, T., Natoli, C. and Gramoli, V. [2018]. Evaluating the red belly blockchain, *CoRR* **abs/1812.11747**.
URL: *http://arxiv.org/abs/1812.11747*

[48] *Cypherpunks Manifesto* [1993]. https://nakamotoinstitute.org/static/docs/cypherpunk-manifesto.txt.

[49] Dang, S. [2021]. Maintenance error caused facebook's 6-hour outage, company says.
URL: *https://www.reuters.com/technology/facebook-says-maintenance-error-caused-mondays-6-hour-outage-2021-10-05/*

[50] Delporte-Gallet, C., Fauconnier, H. and Guerraoui, R. [2002]. A realistic look at failure detectors, *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, IEEE Computer Society, USA, pp. 345–353.

[51] Desmedt, Y. [1988]. Society and group oriented cryptography: a new concept, *in* C. Pomerance (ed.), *Advances in Cryptology — CRYPTO '87*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 120–127.

[52] Desmedt, Y. and Frankel, Y. [1990]. Threshold cryptosystems, *in* G. Brassard (ed.), *Advances in Cryptology — CRYPTO' 89 Proceedings*, Springer New York, New York, NY, pp. 307–315.

[53] Diekert, V. and Rozenberg, G. (eds) [1995]. *The Book of Traces*, World Scientific.
URL: *https://doi.org/10.1142/2563*

[54] Dolev, D. and Reischuk, R. [1985]. Bounds on information exchange for byzantine agreement, *J. ACM* **32**(1): 191–204.
URL: *https://doi.org/10.1145/2455.214112*

[55] Dolev, D. and Strong, H. R. [1982]. Polynomial algorithms for multiple processor agreement, *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, Association for Computing Machinery, New York, NY, USA, pp. 401–407.
URL: *https://doi.org/10.1145/800070.802215*

[56] Douceur, J. J. [2002]. The sybil attack, *Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS)*.
URL: *https://www.microsoft.com/en-us/research/publication/the-sybil-attack/*

[57] Du, J., Sciascia, D., Elnikety, S., Zwaenepoel, W. and Pedone, F. [2014]. Clock-RSM: Low-Latency Inter-datacenter State Machine Replication Using Loosely Synchronized Physical Clocks, *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pp. 343–354.
URL: *https://doi.org/10.1109/DSN.2014.42*

[58] Dwork, C., Lynch, N. and Stockmeyer, L. [1988]. Consensus in the presence of partial synchrony, *J. ACM* **35**(2): 288–323.
URL: *https://doi.org/10.1145/42282.42283*

[59] Dwork, C. and Naor, M. [1993]. Pricing via processing or combatting junk mail, *in* E. F. Brickell (ed.), *Advances in Cryptology — CRYPTO' 92*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 139–147.

[60] Eischer, M. and Distler, T. [2021]. Egalitarian byzantine fault tolerance.

[61] Enes, V., Baquero, C., Gotsman, A. and Sutra, P. [2021]. Efficient replication via timestamp stability, *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, Association for Computing Machinery, New York, NY, USA, pp. 178–193.
URL: *https://doi.org/10.1145/3447786.3456236*

[62] Enes, V., Baquero, C., Rezende, T. F., Gotsman, A., Perrin, M. and Sutra, P. [2020a]. State-machine replication for planet-scale systems, *Proceedings of*

the Fifteenth European Conference on Computer Systems, EuroSys '20, Association for Computing Machinery, New York, NY, USA.
URL: *https://doi.org/10.1145/3342195.3387543*

[63] Enes, V., Baquero, C., Rezende, T. F., Gotsman, A., Perrin, M. and Sutra, P. [2020b]. State-machine replication for planet-scale systems (extended version), *CoRR* **abs/2003.11789**.
URL: *https://arxiv.org/abs/2003.11789*

[64] Fischer, M. J., Lynch, N. A. and Paterson, M. S. [1985]. Impossibility of distributed consensus with one faulty process, *J. ACM* **32**(2): 374–382.
URL: *http://doi.acm.org/10.1145/3149.214121*

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

[66] Garay, J., Kiayias, A. and Leonardos, N. [2015]. The bitcoin backbone protocol: Analysis and applications, *in* E. Oswald and M. Fischlin (eds), *Advances in Cryptology - EUROCRYPT 2015*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 281–310.

[67] Garcia-Molina, H. and Barbara, D. [1985]. How to assign votes in a distributed system, *J. ACM* **32**(4): 841–860.
URL: *https://doi.org/10.1145/4221.4223*

[68] García-Pérez, Á. and Gotsman, A. [2018]. Federated byzantine quorum systems (extended version), *CoRR* **abs/1811.03642**.
URL: *http://arxiv.org/abs/1811.03642*

[69] Gifford, D. K. [1979]. Weighted voting for replicated data, *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP '79, Association for Computing Machinery, New York, NY, USA, pp. 150–162.
URL: *https://doi.org/10.1145/800215.806583*

[70] Gilad, Y., Hemo, R., Micali, S., Vlachos, G. and Zeldovich, N. [2017]. Algorand: Scaling byzantine agreements for cryptocurrencies, *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, ACM, pp. 51–68.
URL: *https://doi.org/10.1145/3132747.3132757*

[71] Gilbert, S. and Lynch, N. [2002]. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, *SIGACT News* **33**(2): 51–59.
URL: *http://doi.acm.org/10.1145/564585.564601*

[72] Girault, A., Gößler, G., Guerraoui, R., Hamza, J. and Seredinschi, D. [2017]. Why you can't beat blockchains: Consistency and high availability in distributed systems, *CoRR* **abs/1710.09209**.
URL: *http://arxiv.org/abs/1710.09209*

[73] Girault, A., Gössler, G., Guerraoui, R., Hamza, J. and Seredinschi, D.-A. [2018]. Monotonic Prefix Consistency in Distributed Systems, *in* C. Baier and L. Caires (eds), *FORTE 2018 - 38th International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, Vol. LNCS-10854 of *Formal Techniques for Distributed Objects, Components, and Systems*, Springer International Publishing, Madrid, Spain, pp. 41–57.
URL: *https://hal.inria.fr/hal-01824817*

[74] Golan Gueta, G., Abraham, I., Grossman, S., Malkhi, D., Pinkas, B., Reiter, M., Seredinschi, D., Tamir, O. and Tomescu, A. [2019]. Sbft: A scalable and decentralized trust infrastructure, *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 568–580.

[75] Goldreich, O., Goldwasser, S. and Micali, S. [1986]. How to construct random functions, *J. ACM* **33**(4): 792–807.
URL: *https://doi.org/10.1145/6490.6503*

[76] Goldwasser, S., Micali, S. and Rivest, R. L. [1988]. A digital signature scheme secure against adaptive chosen-message attacks, *SIAM J. Comput.* **17**(2): 281–308.
URL: *https://doi.org/10.1137/0217017*

[77] Gray, J. and Lamport, L. [2006]. Consensus on transaction commit, *ACM Trans. Database Syst.* **31**(1): 133–160.
URL: *https://doi.org/10.1145/1132863.1132867*

[78] Herlihy, M. [1986]. A quorum-consensus replication method for abstract data types, *ACM Trans. Comput. Syst.* **4**(1): 32–53.
URL: *https://doi.org/10.1145/6306.6308*

[79] Herlihy, M. and Moir, M. [2016]. Enhancing accountability and trust in distributed ledgers, *CoRR* **abs/1606.07490**.
URL: *http://arxiv.org/abs/1606.07490*

[80] Herlihy, M. P. and Wing, J. M. [1990]. Linearizability: A correctness condition for concurrent objects, *ACM Trans. Program. Lang. Syst.* **12**(3): 463–492.
URL: *https://doi.org/10.1145/78969.78972*

[81] Hurfin, M. and Raynal, M. [1999]. A simple and fast asynchronous consensus protocol based on a weak failure detector, *Distributed Computing* **12**(4): 209–223.
URL: *https://doi.org/10.1007/s004460050067*

[82] Israeli, A. and Li, M. [1987]. Bounded time-stamps (extended abstract), *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, IEEE Computer Society, pp. 371–382.
URL: *https://doi.org/10.1109/SFCS.1987.10*

[83] Ittai Abraham, D. M. [2017]. The blockchain consensus layer and bft, *Bulletin of EATCS* .
URL: *http://eatcs.org/beatcs/index.php/beatcs/article/download/506/495*

[84] Katz, J. and Koo, C.-Y. [2009]. On expected constant-round protocols for byzantine agreement, *Journal of Computer and System Sciences* **75**(2): 91–112.
URL: *https://www.sciencedirect.com/science/article/pii/S0022000008000718*

[85] Kokoris-Kogias, E., Jovanovic, P., Gailly, N., Khoffi, I., Gasser, L. and Ford, B. [2016]. Enhancing bitcoin security and performance with strong consistency via collective signing, *CoRR* **abs/1602.06997**.
URL: *http://arxiv.org/abs/1602.06997*

[86] Kuznetsov, P., Tonkikh, A. and Zhang, Y. X. [2021]. Revisiting optimal resilience of fast byzantine consensus, *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, Association for Computing Machinery, New York, NY, USA, pp. 343–353.
URL: *https://doi.org/10.1145/3465084.3467924*

[87] Lamport, L. [1978]. Time, clocks, and the ordering of events in a distributed system, *Commun. ACM* **21**(7): 558–565.
URL: *http://doi.acm.org/10.1145/359545.359563*

[88] Lamport, L. [1998]. The part-time parliament, *ACM Trans. Comput. Syst.* **16**(2): 133–169.
URL: *https://doi.org/10.1145/279227.279229*

[89] Lamport, L. [2001]. Paxos made simple, *ACM SIGACT News (Distributed Computing Column)* **32**(4): 18–25.

[90] Lamport, L. [2005]. Generalized consensus and Paxos, *Technical Report MSR-TR-2005-33*, Microsoft Research.

[91] Lamport, L. [2006a]. Fast paxos, *Distributed Computing* **19**(2): 79–103.

[92] Lamport, L. [2006b]. Lower bounds for asynchronous consensus, *Distributed Computing* **19**(2): 104–125.

[93] Lamport, L., Shostak, R. and Pease, M. [1982]. The byzantine generals problem, *ACM Trans. Program. Lang. Syst.* **4**(3): 382–401.
URL: *https://doi.org/10.1145/357172.357176*

[94] Lepore, C., Ceria, M., Visconti, A., Rao, U. P., Shah, K. A. and Zanolini, L. [2020]. A survey on blockchain consensus with a performance comparison of pow, pos and pure pos, *Mathematics* **8**(10).
URL: *https://www.mdpi.com/2227-7390/8/10/1782*

[95] Li, P., Wang, G., Chen, X., Long, F. and Xu, W. [2020]. Gosig: A scalable and high-performance byzantine consensus for consortium blockchains, *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, Association for Computing Machinery, New York, NY, USA, p. 223–237.
URL: *https://doi.org/10.1145/3419111.3421272*

[96] Lipton, R. J. [1975]. Reduction: A method of proving properties of parallel programs, *Commun. ACM* **18**(12): 717–721.
URL: *https://doi.org/10.1145/361227.361234*

[97] Liu, S., Viotti, P., Cachin, C., Quema, V. and Vukolic, M. [2016]. XFT: Practical fault tolerance beyond crashes, *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, USENIX Association, Savannah, GA, pp. 485–500.
URL: *https://www.usenix.org/conference/osdi16/technical-sessions/presentation/liu*

[98] Lokhava, M., Losa, G., Mazières, D., Hoare, G., Barry, N., Gafni, E., Jove, J., Malinowsky, R. and McCaleb, J. [2019]. Fast and secure global payments with stellar, *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, Association for Computing Machinery, New York, NY, USA, pp. 80–96.
URL: *https://doi.org/10.1145/3341301.3359636*

[99] Lu, Y., Lu, Z., Tang, Q. and Wang, G. [2020]. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited, *Proceedings of the 39th Symposium on Principles of Distributed Computing*, PODC '20, Association for Computing Machinery, New York, NY, USA, pp. 129–138.
URL: *https://doi.org/10.1145/3382734.3405707*

[100] Lynch, N. A. [1996]. *Distributed Algorithms*, Morgan Kaufmann.

[101] Malkhi, D. and Reiter, M. K. [1998]. Byzantine quorum systems, *Distributed Comput.* **11**(4): 203–213.
URL: *https://doi.org/10.1007/s004460050050*

[102] Malkhi, D., Reiter, M. K. and Wool, A. [2000]. The load and availability of byzantine quorum systems, *SIAM Journal on Computing* **29**(6): 1889–1906.
URL: *https://doi.org/10.1137/S0097539797325235*

[103] Malkhi, D. and Terry, D. [2005]. Concise version vectors in winfs, *Proceedings of the 19th International Conference on Distributed Computing*, DISC'05, Springer-Verlag, Berlin, Heidelberg, p. 339–353.

[104] Mao, Y., Junqueira, F. P. and Marzullo, K. [2008]. Mencius: Building efficient replicated state machines for wans, *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 369–384.

[105] Micali, S., Vadhan, S. and Rabin, M. [1999]. Verifiable random functions, *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, IEEE Computer Society, USA, p. 120.

[106] Moraru, I., Andersen, D. G. and Kaminsky, M. [2013]. There is more consensus in egalitarian parliaments, *ACM Symposium on Operating Systems Principles (SOSP)*, pp. 358–372.

[107] Nakamoto, S. [2009]. Bitcoin: A peer-to-peer electronic cash system.
URL: *http://www.bitcoin.org/bitcoin.pdf*

[108] Neiheiser, R., Matos, M. and Rodrigues, L. E. T. [2021]. The quest for scaling BFT consensus through tree-based vote aggregation, *CoRR* **abs/2103.12112**.
URL: *https://arxiv.org/abs/2103.12112*

[109] Ongaro, D. and Ousterhout, J. [2014]. In search of an understandable consensus algorithm, *USENIX Annual Technical Conference (USENIX ATC)*, pp. 305–320.

[110] Pass, R., Seeman, L. and Shelat, A. [2017]. Analysis of the blockchain protocol in asynchronous networks, *in* J.-S. Coron and J. B. Nielsen (eds), *Advances in Cryptology – EUROCRYPT 2017*, Springer International Publishing, Cham, pp. 643–673.

[111] Pease, M., Shostak, R. and Lamport, L. [1980]. Reaching agreement in the presence of faults, *J. ACM* **27**(2): 228–234.
URL: *https://doi.org/10.1145/322186.322188*

[112] Pedone, F. and Schiper, A. [1999]. Generic broadcast, *International Symposium on Distributed Computing (DISC)*, pp. 94–108.

[113] Platform, G. C. [2021]. Google cloud datastore incident 19006.
URL: *https://status.cloud.google.com/incident/cloud-datastore/19006*

[114] Ranchal-Pedrosa, A. and Gramoli, V. [2020]. Blockchain is dead, long live blockchain! accountable state machine replication for longlasting blockchain, *CoRR* **abs/2007.10541**.
URL: *https://arxiv.org/abs/2007.10541*

[115] Rezende, T. F. [2021]. Scalable byzantine fault tolerance through leaderless state machine replication, *2nd International Workshop on Foundations of Consensus and Distributed Ledgers*.
URL: *https://www.discotec.org/2021/focodile#scalable-byzantine-fault-tolerance-through-leaderless-state-machine-replication*

[116] Rezende, T. F. and Sutra, P. [2020]. Leaderless state-machine replication: Specification, properties, limits, *in* H. Attiya (ed.), *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, Vol. 179 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 24:1–24:17.
URL: *https://doi.org/10.4230/LIPIcs.DISC.2020.24*

[117] Rezende, T. F., Sutra, P., Saramago, R. Q. and Camargos, L. [2017]. On making Generalized Paxos practical, *AINA 2017 : 31st IEEE International Conference on Advanced Information Networking and Applications*, IEEE Computer Society, Taipei, Taiwan, pp. 347 – 354.
URL: *https://hal.archives-ouvertes.fr/hal-01576869*

[118] Rocket, T., Yin, M., Sekniqi, K., van Renesse, R. and Sirer, E. G. [2019]. Scalable and probabilistic leaderless BFT consensus through metastability, *CoRR* **abs/1906.08936**.
URL: *http://arxiv.org/abs/1906.08936*

[119] Santiago, C., Ren, S., Lee, C. and Ryu, M. [2021]. Concordia: A streamlined consensus protocol for blockchain networks, *IEEE Access* **9**: 13173–13185.

[120] Schiper, A. [1997]. Early consensus in an asynchronous system with a weak failure detector, *Distrib. Comput.* **10**(3): 149–157.
URL: *https://doi.org/10.1007/s004460050032*

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

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

[123] Shoker, A. [2012]. *PhD Thesis: BYZANTINE FAULT TOLERANCE: FROM STATIC SELECTION TO DYNAMIC SWITCHING*, Theses, Université Paul Sabatier - Toulouse III. Thesis with European Label.
URL: *https://tel.archives-ouvertes.fr/tel-00790443*

[124] Sonnino, A. and Danezis, G. [2019]. Sybilquorum: Open distributed ledgers through trust networks, *CoRR* **abs/1906.12237**.
URL: *http://arxiv.org/abs/1906.12237*

[125] Sousa, J. a. and Bessani, A. [2012]. From byzantine consensus to bft state machine replication: A latency-optimal transformation, *Proceedings of the 2012 Ninth European Dependable Computing Conference*, EDCC '12, IEEE Computer Society, USA, pp. 37–48.
URL: *https://doi.org/10.1109/EDCC.2012.32*

[126] Sousa, J., Bessani, A. and Vukolic, M. [2018]. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform, *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 51–58.

[127] Sutra, P. and Shapiro, M. [2011]. Fast genuine generalized consensus, *IEEE Symposium on Reliable Distributed Systems (SRDS)*, pp. 255–264.

[128] Sweney, M. [2021]. Facebook outage highlights global over-reliance on its services.
URL: *https://www.theguardian.com/technology/2021/oct/05/facebook-outage-highlights-global-over-reliance-on-its-services*

[129] Syta, E., Tamas, I., Visher, D., Wolinsky, D. I., Jovanovic, P., Gasser, L., Gailly, N., Khoffi, I. and Ford, B. [2016]. Keeping authorities "honest or bust" with decentralized witness cosigning.

[130] Toumlilt, I., Sutra, P. and Shapiro, M. [2021]. Highly-Available and Consistent Group Collaboration at the Edge with Colony, *Middleware 2021: 22nd International Middleware Conference*, ACM, Québec (online), Canada.
URL: *https://hal.inria.fr/hal-03353663*

[131] Turcu, A., Peluso, S., Palmieri, R. and Ravindran, B. [2014]. Be general and don't give up consistency in geo-replicated transactional systems, *International Conference on Principles of Distributed Systems (OPODIS)*, pp. 33–48.

[132] V. Enes, T. Rezende, A. Gotsman, M. Perrin, P. Sutra [2018]. Fast state-machine replication via monotonic generic broadcast. Available at `https://sites.google.com/site/mgbextended/paper.pdf`.
URL: *https://sites.google.com/site/mgbextended/paper.pdf*

[133] Veronese, G. S., Correia, M., Bessani, A. N. and Lung, L. C. [2009]. Spin one's wheels? byzantine fault tolerance with a spinning primary, *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, SRDS '09, IEEE Computer Society, USA, pp. 135–144.
URL: *https://doi.org/10.1109/SRDS.2009.36*

[134] Veronese, G. S., Correia, M., Bessani, A. N., Lung, L. C. and Verissimo, P. [2013]. Efficient byzantine fault-tolerance, *IEEE Transactions on Computers* **62**(1): 16–30.

[135] *Weak Subjectivity* [2014]. https://blog.ethereum.org/2014/11/25/proof-stake-learned-love-weak-subjectivity/.

[136] Whittaker, M., Ailijiang, A., Charapko, A., Demirbas, M., Giridharan, N., Hellerstein, J. M., Howard, H., Stoica, I. and Szekeres, A. [2020]. Scaling replicated state machines with compartmentalization [technical report], *CoRR* **abs/2012.15762**.
URL: *https://arxiv.org/abs/2012.15762*

[137] Whittaker, M., Giridharan, N., Szekeres, A., Hellerstein, J. M. and Stoica, I. [n.d.]. "bipartisan paxos: A family of fast, leaderless, modular state machine replication protocols". preprint on webpage at `https://mwhittaker.github.io/publications/bipartisan_paxos.pdf`.

[138] *Year 2000 problem* [2021]. https://en.wikipedia.org/wiki/Year_2000_problem.

[139] Yin, M., Malkhi, D., Reiter, M. K., Gueta, G. G. and Abraham, I. [2019]. Hotstuff: Bft consensus with linearity and responsiveness, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, Association for Computing Machinery, New York, NY, USA, pp. 347–356.
URL: *https://doi.org/10.1145/3293611.3331591*

[140] Zhang, Y., Setty, S., Chen, Q., Zhou, L. and Alvisi, L. [2020]. Byzantine ordered consensus without byzantine oligarchy, *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, pp. 633–649.
URL: *https://www.usenix.org/conference/osdi20/presentation/zhang-yunhao*

[141] Zieliński, P. [2005]. Optimistic generic broadcast, *in* P. Fraigniaud (ed.), *Distributed Computing*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 369–383.

Résumé en Français

Les services internet modernes sont déployés sur une infrastructure en constante expansion composée de plusieurs centres de données comprenant chacun des milliers d'ordinateurs, souvent répartis dans le monde entier. Nos sociétés étant de plus en plus dépendantes de ces services, leur indisponibilité a des conséquences très étendues. Autrefois limitées aux coûts monétaires pour leurs propriétaires, les défaillances de services affectent maintenant les entreprises et les communications dans le monde entier. Cette thèse vise à faire avancer l'état de l'art sur les techniques utilisées pour construire des services distribués hautement disponibles en fournissant de nouvelles abstractions et protocoles qui prennent en compte ce nouveau paradigme à l'échelle planétaire.

Contexte

Pour atteindre une haute disponibilité, les services distribués répliquent souvent leurs données critiques dans plusieurs répliques. Ainsi, en cas de panne, le service reste opérationnel car les clients peuvent toujours y accéder par le biais d'autres répliques en état de marche. Dans les systèmes distribués, la technique classique pour mettre en œuvre de tels services tolérants aux pannes est appelée *State-Machine Replication (SMR)* [122]. Elle permet à un ensemble de processus distribués (répliques) de construire un objet partagé linéarisable [80]. Dans SMR, un service est défini comme une machine d'état déterministe avec un ensemble de commandes et chaque processus maintient sa propre copie locale de la machine. Un protocole SMR coordonne l'exécution des commandes appliquées à la machine d'état, garantissant que les répliques restent synchronisées.

Au cœur du protocole SMR se trouve un protocole de consensus utilisé par les répliques pour décider d'un ordre commun d'application des commandes. Ces dernières années, le protocole de consensus le plus utilisé a été Paxos [88], présent dans des systèmes bien établis comme le service de verrouillage Chubby [35] et le magasin distribué Spanner [44]. Malgré son succès auprès des prati-

ciens, Paxos présente des limites bien connues. Notamment, sa conception est centrée sur une réplique leader responsable d'ordonner les commandes de tous les clients. Une telle approche basée sur le leader est présente dans un large éventail de protocoles SMR classiques (par exemple Raft [109]).

Motivation et problèmes de recherche

Le paradigme de réplication des services distribués modernes est de type *géo-réplication*, c'est-à-dire que les répliques sont placées sur plusieurs sites géographiques. La géo-réplication permet une disponibilité accrue et une faible latence, dès lors que les clients peuvent communiquer avec la réplique géographique la plus proche.

En raison de leur dépendance à l'égard d'une réplique leader, les protocoles SMR classiques tels que Paxos et Raft offrent une évolutivité et une disponibilité limitées en cas de géo-réplication (par exemple, les clients géographiquement éloignés de la réplique leader souffriront d'une latence élevée lorsqu'ils interagiront avec le système). Pour résoudre ce problème, des protocoles récents tels que EPaxos [106] et Mencius [104] suivent plutôt une approche *leaderless*, dans laquelle chaque réplique est capable de progresser tant qu'elle peut contacter un sous-ensemble de ses pairs. Bien que cette nouvelle classe de protocoles sans leader offre une approche prometteuse pour aborder la géo-réplication, leur adoption à grande échelle est entravée par leur grande complexité et leur approche *ad-hoc* de l'absence de leader.

Dans le contexte de la géo-réplication, les blockchains sont une application qui a fait l'objet de nombreuses recherches. Une blockchain peut être considérée comme un registre de transactions inviolable, distribué et répliqué. Les transactions entre les clients sont regroupées en blocs, chaque bloc est lié cryptographiquement à un bloc précédent et un protocole de réplication résistant aux panne arbitraires des répliques (panne *Byzantine*) est responsable de la réplication correcte de la chaîne de blocs entre les répliques. Tous les protocoles SMR susmentionnés ont pour caractéristique commune de ne prendre en charge que les panne bénignes des répliques et ne peuvent être utilisés tels quels dans le contexte des blockchains.

Le protocole de chaîne de blocs le plus célèbre et qui a suscité l'intérêt dans ce domaine est sans doute Bitcoin [107]. Le bitcoin suit le modèle *permissionless*, où n'importe qui peut rejoindre le réseau et exécuter une réplique. Garantir la cohérence de la blockchain dans un tel modèle, où les processus peuvent rejoindre et quitter le réseau à tout moment, agir de manière malveillante pour leurs

propres gains financiers et falsifier des identités, est particulièrement complexe et coûteux (il s'appuie sur un protocole de réplication byzantin coûteux en calcul [27]). Pour éviter les complexités du modèle précédent et les coûts qui y sont associés, une alternative consiste à considérer un modèle différent (*permissioned*) où l'appartenance à la blockchain est contrôlée par un ensemble d'entités connues comme des entreprises. Dans ce cas, les blockchains sont souvent alimentées par une réplication byzantine classique (SMR byzantine). Lorsque les protocoles de blockchain avec permission suivent l'approche classique basée sur les leaders, ils souffrent de problèmes d'évolutivité et de disponibilité, de la même manière que leurs homologues non byzantins.

Contributions

Dans cette thèse, nous proposons un *framework* qui capture l'essence de la Réplication de Machine D'état sans leader(Leaderless State-Machine Replication - Leaderless SMR) et nous énonçons formellement certaines de ses limites.

Nous définissons la SMR sans leader et la décomposons en éléments de base (§3.2). De plus, nous introduisons un ensemble de propriétés souhaitables pour le SMR sans leader: (R)eliability, (O)ptimal (L)atency et (L)oad Balancing. Les protocoles qui répondent à toutes les propriétés ROLL sont soumis à un compromis entre performance et fiabilité. Plus précisément, dans un système de n processus, le théorème ROLL (§3.3) stipule que les protocoles SMR sans leader sont soumis à l'inégalité $2F + f - 1 \leq n$, où $n - F$ est la taille du quorum du chemin rapide (*fast path*) et f est le nombre maximal de panne tolérées. Un protocole est ROLL-optimal lorsque F et f ne peuvent pas être améliorés selon cette inégalité. Nous établissons que les protocoles ROLL-optimaux sont soumis à un effet de chaînage qui affecte leurs performances (§3.4). Comme EPaxos est ROLL-optimal et Mencius non, l'effet de chaînage explique les résultats de performance observés dans Figure 2.4. Enfin, nous discutons des implications de ce résultat (§3.5) puis mettons notre travail en perspective (§3.6).

De plus, nous adaptons notre *framework* pour supporter les panne byzantines et présentons le premier *framework* pour la SMR byzantine sans leader. Nous montrons que, lorsqu'il est correctement instancié, il permet de contourner le problème de scalabilité des protocoles SMR byzantins dirigés par des leaders. La première instanciation d'intérêt est une version byzantine d'EPaxos [106]. La seconde instanciation est un nouveau protocole que nous appelons *Wintermute* (§4.3.3). Ce protocole a un comportement asymptotique globalement meilleur que les protocoles SMR BFT traditionnels. Plus précisément, il présente une

charge et une complexité de message inférieures à celles de l'art antérieur, ce qui lui permet de contourner les problèmes d'évolutivité inhérents aux solutions BFT, tels qu'ils se présentent, par exemple, dans le contexte des permissioned blockchains.

ECOLE
DOCTORALE

Titre : Réplication de Machines à états sans leader: des fautes franches aux fautes byzantines

Mots clés : Réplication de Machines à états, Consensus sans leader, Fautes Byzantine, Blockchain

Résumé :
Les services distribués modernes doivent être hautement disponibles, car nos sociétés en sont de plus en plus dépendantes. La manière la plus courante d'obtenir une haute disponibilité est de répliquer les données dans plusieurs répliques du service. De cette façon, le service reste opérationnel en cas de pannes, car les clients peuvent être relayés vers d'autres répliques qui fonctionnent. Dans les systèmes distribués, la technique classique pour mettre en œuvre de tels services tolérants aux pannes est appelée réplication de machine d'état (State-Machine Replication, SMR), où un service est défini comme une machine d'état déterministe et chaque réplique conserve une copie locale de la machine. Pour garantir la cohérence du service, les répliques se coordonnent entre elles et conviennent de l'ordre des transitions à appliquer à leurs copies de la machine d'état.
La réplication effectuée par les services Internet modernes s'étend sur plusieurs lieux géographiques (géo-réplication). Cela permet une disponibilité accrue et une faible latence, puisque les clients peuvent communiquer avec la réplique géographique la plus proche.
En raison de leur dépendance avec une réplique leader, coordonnant les changements de transition, les protocoles SMR classiques offrent une évolutivité et une disponibilité limitées dans ce contexte. Pour résoudre ce problème, les protocoles récents suivent plutôt une approche sans leader, dans laquelle chaque réplique est capable de progresser en utilisant un quorum de ses pairs. Ces nouveaux protocoles sans leader sont complexes et chacun d'entre eux présente une approche ad-hoc de l'absence de leader. La première contribution de cette thèse est un framework qui capture l'essence de SMR sans leader (Leaderless SMR) et la formalisation de certaines de ses limites.
En raison de la nature de plus en plus sensible des services répliqués, l'utilisation de simples pannes bénignes n'est plus suffisante. Les recherches récentes se dirigent vers le développement de protocoles qui supportent le comportement arbitraire de certaines répliques (pannes Byzantines) et qui prospèrent également dans un environnement géo-répliqué. Les blockchains sont un exemple de ce nouveau type de services répliqués sensibles qui a fait l'objet de nombreuses recherches.
Les blockchains permissioned utilisent des protocoles SMR byzantins. Comme ces protocoles utilisent un leader, ils souffrent de problèmes d'évolutivité et de disponibilité, de la même manière que leurs homologues non byzantins. Dans la deuxième partie de cette thèse, nous adaptons notre framework pour supporter les pannes byzantines et présentons le premier framework pour le SMR byzantin sans leader. De plus, nous montrons que lorsqu'il est correctement instancié, il permet de contourner les problèmes de scalabilité dans les protocoles SMR byzantins dirigés par des leaders pour les permissioned blockchains.

Title : Leaderless State-Machine Replication: From Fail-stop to Byzantine Failures

Keywords : State-Machine Replication, Leaderless Consensus, Byzantine Failures, Blockchain

Abstract :
Modern distributed services are expected to be highly available, as our societies have been growing increasingly dependent on them. The common way to achieve high availability is through the replication of data in multiple service replicas. In this way, the service remains operational in case of failures as clients can be relayed to other working replicas.
In distributed systems, the classic technique to implement such fault-tolerant services is called State-Machine Replication (SMR), where a service is defined as a deterministic state-machine and each replica keeps a local copy of the machine. To guarantee that the service remains consistent, replicas coordinate with each other and agree on the order of transitions to be applied to their copies of the state-machine.
The replication performed by modern Internet services spans across several geographical locations (geo-replication). This allows for increased availability and low latency, since clients can communicate with the closest geographical replica. Due to their reliance on a leader replica, classical SMR protocols offer limited scalability and availability under this setting. To solve this problem, recent protocols follow instead a leaderless approach, in which each replica is able to make progress using a quorum of its peers. These new leaderless protocols are complex and each one presents an ad-hoc approach to leaderlessness.
The first contribution of this thesis is a framework that captures the essence of Leaderless State-Machine Replication (Leaderless SMR) and the formalization of some of its limits. Due to the increasingly sensitive nature of replicated services, leveraging simple benign failures is no longer enough. Recent research is headed towards developing protocols that support arbitrary behavior of some replicas (Byzantine failures) and that also thrive in a geo-replicated environment. An example of this new type of sensitive replicated services that has been the focus of a lot of research are blockchains. When such Byzantine protocols follow the classic leader-driven approach they suffer from scalability and availability issues, similarly to their non-byzantine counterparts.
In the second part of this thesis, we adapt our framework to support Byzantine failures and present the first framework for Byzantine Leaderless SMR. Furthermore, we show that when properly instantiated it allows to sidestep the scalability problems in leader-driven Byzantine SMR protocols for permissioned blockchains.

Institut Polytechnique de Paris
91120 Palaiseau, France