UC Santa Cruz
Journal of Systems Research

Title
Fast State-Machine Replication for Byzantine Systems and Blockchain

Permalink
https://escholarship.org/uc/item/1mr7w8rs

Journal
Journal of Systems Research, 6(1)

Authors
Sutra, Pierre
Shprenger, Konstantin
Rezende, Tuanir França

Publication Date
2026-11-15

DOI
10.5070/SR3.70465

Copyright Information
This work is made available under the terms of a Creative Commons Attribution-NonCommercial License, available at https://creativecommons.org/licenses/by-nc/4.0/

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

[SOLUTION] FAST STATE-MACHINE REPLICATION FOR BYZANTINE SYSTEMS AND BLOCKCHAIN

Pierre Sutra
Institut Polytechnique de Paris

Konstantin Shprenger
Institut Polytechnique de Paris

Tuanir França Rezende
CEA

Foreword by the Area Chair

Byzantine state-machine replication allows distributed systems and blockchains to maintain consistent states, even when some machines misbehave arbitrarily. This paper presents a general framework for designing leaderless Byzantine fault-tolerant replication protocols that take advantage of operations that do not conflict with each other. Using this framework, the authors develop a protocol that achieves optimal latency of two message delays under favorable conditions, while exploring different tradeoffs between key metrics such as message complexity and resilience. We hope this work inspires further research on building efficient and reliable distributed systems.

– Roberto Palmieri, Lehigh University
– Lewis Tseng, University of Massachusetts Lowell

Reviewers

- Lewis Tseng, University of Massachusetts Lowell
- Roberto Palmieri, Lehigh University
- Suyash Gupta, University of Oregon

Artifacts

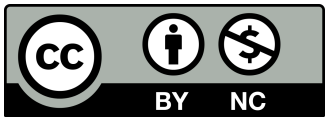
The artifacts for this work were independently evaluated by the Artifact Evaluation Board (AEB) led by Eric Eide.
The AEB determined that the artifact is usable by a third party and that it allows reproducing the main results from the paper.
Artifact: https://anonymous.4open.science/r/blsmr-7B71

Reviews

Anonymized reviews are publicly available.
Reviews: https://zenodo.org/records/24550788/files/Sol_Fast_State_Machine_Replication_Reviews.pdf

Copyright and License

Licensed under Creative Common License CC-BY-NC. Copyright retained by the authors.



FAST STATE-MACHINE REPLICATION FOR BYZANTINE SYSTEMS AND BLOCKCHAIN

PIERRE SUTRA
Institut Polytechnique de Paris

KONSTANTIN SHPRENGER
Institut Polytechnique de Paris

TUANIR FRANÇA REZENDE
CEA

Abstract

State-machine replication (SMR) hardened for Byzantine fault-tolerance (BFT) is a cornerstone of modern distributed systems. In particular, BFT SMR provides the foundation for modern blockchains. Standard BFT SMR protocols rely on a leader replica to order commands submitted by clients to the replicated state machine. Some of these protocols allow a command to execute at the leader after just two message delays. Another class of protocols is leaderless, permitting each replica to progress as long as it contacts enough of its peers. Leaderless protocols distribute the client load evenly and avoid the potential disruptions caused by a Byzantine leader. They can also leverage application-level commutativity for better performance. Unfortunately, to date, existing leaderless solutions take more than two message delays to order a command, or they slow down as soon as a single pair of non-commuting commands is detected. In this paper, we address this limitation. We present the first fast BFT SMR protocol. This protocol executes conflict-free commands after a single round-trip from any replica in the system. The protocol is explained with the help of a general framework to create correct-by-design leaderless BFT SMR protocols. To demonstrate the utility of the framework, we also present a Byzantine fault-tolerant variation of Egalitarian Paxos and a leaderless protocol tailored for large-scale deployments. Finally, discrete-event simulations with up to a thousand replicas show that our protocols scale and that the fast one achieves a 2–5× lower median latency than the state of the art.

1 Introduction

Context. Blockchain is a recent technology to construct robust decentralized storage and applications. At its core, this technology is based on a distributed Byzantine fault-tolerant (BFT) replication protocol. Early blockchain designs relied on proof-of-work (PoW) to perform data replication [20, 76]. This approach offers a high level of security but it also has major drawbacks, being energy-hungry and slow. For that reason, modern blockchains instead employ proof-of-stake (PoS) [8]. Internally, several PoS protocols use state-machine replication (SMR) [19, 24, 34, 46, 100]. These protocols are inherited from decades of scientific advances in the field of distributed computing. They deliver better throughput and latency than asymptotic PoW mechanisms and provide linearizable operations. Both properties greatly simplify the task of programming blockchain applications. In this paper, we consider permissioned systems, where the set of replicas is fixed and known in advance, as in consortium blockchains or in the committees of PoS protocols.

Motivation. In the best case, a state-machine command executes after two message delays, that is, one round-trip to the replicas [63]. The design of such BFT SMR protocols is the subject of extensive research [1, 6, 48, 58]. Because SMR reduces to consensus, this effort also exists for agreement protocols [2, 59, 60, 72]. Most of the solutions rely on a central leader replica. If a command is not submitted at the leader replica, its execution requires waiting for additional message delays.

Another class of SMR protocols does not employ a stable leader but instead decentralizes the task of ordering commands. These solutions are generally referred to as *leaderless* [33]. Leaderless SMR permits load distribution and avoids potential disruptions due to a Byzantine leader. A common design is to rotate consensus instances [95]. Another approach is the use of a DAG data structure [7, 56, 89].

Leaderless protocols submit commands from multiple replicas at the same time. A key observation is that concurrent commands that commute (conflict-free) need not be ordered to reach linearizability [62]. Hence, to avoid ordering commands, some leaderless protocols [1, 5, 10, 41, 79] leverage commutativity at the application level. As concurrent commands are often conflict-free in practice, this reduces latency in the common case.

Unfortunately, *none* of the existing leaderless solutions permits executing a conflict-free command after two message delays (see Table 1). ezBFT [5] claimed it was the first such protocol but it is actually flawed [84]. ISOS [41] requires three message delays to execute a command in the fast path. Q/U [1] is practical only when a unique client performs non-commuting operations. When contention happens, the protocol resorts to an exponential back-off mechanism which impairs its usage [6]. The solution in [79] slows down as soon as a single pair of non-commuting commands is detected. Byblos [10] leverages commutativity at execution. However, the protocol offers a single (slow) path which requires five message delays.

Contributions. To date, no BFT SMR protocol is *fast*, that is, allows a conflict-free command to execute in two message delays from any replica. This work fills this gap. It presents the first protocol that offers such a property.

The protocol is specified with the help of a general framework that implements Byzantine leaderless state-machine replication (BLSMR). The framework is an abstract distributed machine which is of interest on its own. It provides a principled approach to construct correct-by-design protocols—a difficult task as several algorithms were found erroneous [2, 84, 85, 91].

At a high level, the abstract machine works as follows. It uses a directed graph to encode execution constraints, or *dependencies*, between non-commuting commands. To build the graph, the machine calls two services: a dependency discovery service (DDS) and a consensus service. Initially, a command is submitted at some replica that is in charge of coordinating its execution. The coordinator replica first accesses the DDS service to discover its dependencies. Then, it calls consensus to commit them. At a replica, a command executes once its dependencies are committed and have already been executed. The machine also offers a fast path to skip consensus. A fast protocol implements this mechanism: in the fast path, a command executes at its coordinator replica after a single round-trip to a nearby quorum.

In total, three protocols are implemented using the framework (see Table 1). They are alternative instantiations of the abstract machine. The first protocol is a Byzantine-hardened variation of Egalitarian Paxos [75]. This protocol is pedagogical and explains how to instantiate the abstract distributed machine in simple terms. The second protocol, Wintermute, is our fast solution. It implements the fast path using $n \geq 5f+1$ replicas. To the best of our knowledge, Wintermute is the first BFT SMR protocol to be fast. The third protocol, 3Jane, lowers the load of the system, that is, the worst-case probability a process is chosen to order a command. In a large-scale deployment, it allows each command to be ordered by a small fraction of the available machines.

We evaluate our protocols in a discrete-event simulator against state-of-the-art solutions. Our evaluation campaign reveals that *(i)* in simulation and depending on the system topology, Wintermute exhibits a 2–5× lower median latency than the competition; and *(ii)* with a low number of failures ($f \leq 10$ with 1,024 replicas), 3Jane achieves the best scalability while distributing the load evenly.

Outline. The remainder of this paper is organized as follows: §2 details our system model and the problem at hand. §3 introduces leaderless Byzantine state-machine replication (BLSMR). The abstract BLSMR machine is presented in §4. The three protocols implementing the machine are detailed in §5, and evaluated in §6. We survey related work in §7, before closing in §8.

Our algorithms and abstractions are specified in TLA$^+$ and model-checked on small instances for crash faults (see "Artifacts"). The Byzantine case is established by the proofs in the Appendix.

	Metrics					Leaderless	Commut.
Protocol	$\mathcal{L}$	$\mathcal{M}$	$\mathcal{A}$	$\mathcal{D}$	$\mathcal{R}$		
PBFT [25]	3+1	$O(n^2)$	$O(n^2)$	1	$f < n/3$	×	×
FAB Paxos [72]	2+1	$O(n^2)$	$O(n^2)$	1	$f < n/5$	×	×
Quorum [6]	5	$O(n)$	$O(n)$	1	$f < n/3$	√	×
Tendermint [19]	3	$O(n^2)$	$O(n^2)$	1	$f < n/3$	√	×
HotStuff-1 [55]	3	$O(n)$	$O(n)$	1	$f < n/3$	√	×
DAG-Rider [56]	12	$O(n^3)$	$O(n^3)$	1	$f < n/3$	√	×
Bullshark [89]	9	$O(n^2)$	$O(n^2)$	1	$f < n/3$	√	×
Mysticeti [7]	3	$O(n^2)$	$O(n^2)$	1	$f < n/3$	√	√
PGB [79]	5	$O(n^2)$	$O(n)$	1	$f < n/5$	√	√
Byblos [10]	5	$O(n^2)$	$O(n^2)$	1	$f < n/4$	√	√
ISOS [41]	3	$O(n^2)$	$O(n^2)$	1	$f < n/3$	√	√
à la EPaxos (§5.1)	5	$O(n^2)$	$O(n^2)$	1	$f < n/3$	√	√
Wintermute (§5.2)	**2**	$O(n)$	$O(n)$	1	$f < n/5$	√	√
3Jane (§5.3)	4	$O(\sqrt{fn})$	$O(f\sqrt{fn})$	$O(\sqrt{f/n})$	$f \leq \frac{\sqrt{n}-2}{3}$	√	√

Table 1: Comparing BFT SMR protocols in a system of n processes with at most f failures. The metrics are: conflict-free latency ($\mathcal{L}$), message complexity ($\mathcal{M}$), authenticator complexity ($\mathcal{A}$), load ($\mathcal{D}$), and resilience ($\mathcal{R}$). Leaderless protocols (√) achieve the same latency everywhere, while leader-driven protocols (×) need an additional message delay (+1) for non-leader processes. Commutativity is ignored (×), partially leveraged (√), or leveraged (√); details in §5.4.

2 Preliminaries

2.1 System model

We follow the standard model in PBFT [25]. The system consists of a set $\mathbb{P}$ of distributed processes, with $n = |\mathbb{P}|$. This set is static and known in advance by all the processes, which is commonly referred to as the permissioned setting. We discuss the permissionless setting in §8. Processes may deviate arbitrarily from the protocol. If a process deviates, we say that it is faulty, or *Byzantine*. Otherwise, the process is *correct*. In a run, at most $f < n/3$ failures may happen [65]. The bound on f is protocol-specific: some of the protocols presented in this paper require a stronger assumption (e.g., $f < n/5$ for Wintermute), as stated where appropriate.

Processes communicate via message passing and any two processes are connected by a link. Links are reliable, that is, if a correct process p sends a message m to a correct process q, eventually q receives m (provided it executes enough receptions). Byzantine processes can spy on links connecting any two processes. The system is partially synchronous [40]: after some (unknown) global stabilization time (GST), messages take at most Δ units of time to reach their destinations.

We assume the existence of collision-resistant hash functions, digital signatures, and a public-key infrastructure (PKI). In particular, any correct replica can authenticate messages it sends by signing them. Faulty processes are computationally bounded and cannot break these cryptographic techniques.

2.2 State-machine replication

State-machine replication (SMR) is the common approach to implement fault-tolerant distributed services in such systems. In this approach, a service is defined by a deterministic state machine together with a set of commands (C). Byzantine processes cannot forge the state-machine commands. Each process holds a local copy of the state machine. For some command c, *coord*(c) denotes its *coordinator*. This process is in charge of submitting command c to the service on behalf of an external client. As in [100], we omit clients from most of the discussion, and defer to the standard literature for command numbering, re-submission, and de-duplication.

A service implemented with SMR is *linearizable* [28, 52]. This means that commands appear as if executed sequentially on a single copy of the state machine in an order consistent with the real-time order, i.e., the order of non-overlapping command invocations. As observed in [62, 77], for the replicated service to be linearizable, the SMR protocol does not need to ensure that commands are executed at processes in the exact same order: it is enough to agree on the order of non-commuting commands. Formally, two commands c and c' *commute* if in every state s of the state machine: *(i)* executing c followed by c' or c' followed by c in s leads to the same state; and *(ii)* c returns the same response in s as in the state obtained by executing c' in s, and vice versa. We say that c and c' *conflict*, written $\mathsf{c} \asymp \mathsf{c}'$, when they may not commute; in practice, $\asymp$ over-approximates non-commutation (§3.1). Then, the specification of the SMR protocol sufficient for linearizability is given by the following properties:

Validity: If a correct process executes c then c is a command.

Integrity: A correct process executes each command at most once.

Consistency: Correct processes execute conflicting commands in the same order.

Liveness: If a command is submitted by a correct process or executed at some correct process, then it is eventually executed at all correct processes.

We next define a class of SMR protocols that can execute commands fast under favorable conditions. One such condition is that the system is synchronous and failure-free in the following sense. Recall that Δ is an upper bound on the maximal time it takes for a message to reach its destination. We say that the events that happen during the time interval $[0, \Delta)$ form *the first round*, the events that happen during the time interval $[\Delta, 2\Delta)$ *the second round*, and so on.

Definition 1. A run is **failure-free synchronous** if:

- All processes are correct.
- All messages sent during a round are delivered at the beginning of the next round, and each process receives all of them before taking any other step.
- A message a process sends to itself is received immediately.
- All local computations are instantaneous.

Another condition for a command to be executed fast is that there are no concurrent conflicting commands, as defined below.

Definition 2. We say that c **precedes** c' in a run if at the time c' is submitted, all the correct processes have already executed c. Commands c and c' are **concurrent** when neither precedes the other. A command c is **conflict-free** if it does not conflict with any concurrent command.

Using the above definitions, we are now able to specify what it means for a BFT SMR protocol to be fast.

Definition 3. A run of an SMR protocol is **fast** when any conflict-free command submitted at time t by a process p gets executed at p by time $t + 2\Delta$. A protocol is **fast** if every failure-free synchronous run of the protocol is fast.

Fast protocols execute conflict-free commands after two message delays. When these commands have no return values, such as transactions in blockchain, this is sufficient for the client to return: the proof that permitted the command to execute locally contains enough information. This base observation strongly motivates the design of such protocols.

2.3 Problem statement

We are interested in the following problem: *Is it possible to implement a fast Byzantine fault-tolerant state-machine replication protocol?* This paper provides an affirmative answer to that question. To construct a matching protocol, we resort to an approach in which there is no leader replica. In some sense, this is needed in the definition above since *any* process can execute a conflict-free command in two message delays. The next section presents a general framework inspired by Egalitarian Paxos [75] to specify such leaderless protocols. Further, we present an abstract machine that implements this framework and explain how to instantiate it into a fast protocol.

3 Byzantine Leaderless State-Machine Replication

To order commands, one approach is to route them through a distinguished leader process. This design is standard but increases latency at non-leader processes. Leaderless protocols instead decentralize the task of ordering commands. Many leaderless SMR protocols exist [10, 56, 71, 75, 79, 98, 102]. Below, we extend the work of Moraru, Andersen and Kaminsky [75] on Egalitarian Paxos to Byzantine systems.

3.1 Principles

A central notion in [75] is the *dependency graph*. This graph records how commands execute against the copy of the state machine. As observed in [62,77], commuting commands may execute in any order. As a consequence, they do not need to be adjacent in the dependency graph. For instance, consider that the replicated service stores a table T of bank accounts. Commands $\mathsf{b} := T[\text{Bob}] \leftarrow 42\mathbf{\upsilon}$ and $\mathsf{c} := T[\text{Alice}] \leftarrow 5\mathbf{\upsilon}$ respectively assign $42\mathbf{\upsilon}$ and $5\mathbf{\upsilon}$ to the accounts of Bob and Alice (for some token $\mathbf{\upsilon}$). These two commands commute because they have no response value and executing them in any order leads to the same state for table T. Command b does not commute with $\mathsf{a} := T[\text{Bob}]$ that returns the balance of Bob.

In practice, commutativity is over-approximated with the notion of *conflicts*. As mentioned earlier, we denote by $\asymp$ the binary, non-reflexive, and symmetric relation over $\mathcal{C}$ that captures conflicts. In the above example, a single pair of commands conflicts: $\mathsf{a} \asymp \mathsf{b}$.

Each process maintains a local dependency graph to compute the order in which it executes the state-machine commands. For a command c, the incoming neighbors of c in the graph are its *dependencies*, and they should execute before (or jointly with) c. At a process, the dependency graph is encoded in the *deps* map. This map stores a relation from $\mathcal{C}$ to $2^{\mathcal{C}} \cup \{\bot\}$. For a command c, $\mathit{phase}(\mathsf{c})$ defines four possible values: start, pending, commit and stable. Each phase corresponds to a predicate over *deps*. In detail, when c is not in *deps*, it is in the start phase, the initial phase of every command. Command c is pending when $\mathit{deps}(\mathsf{c})$ equals $\bot$. This happens upon calling $\mathit{submit}(\mathsf{c})$ locally and when c is in the start phase. A command can be submitted by one or more processes. A call to $\mathit{commit}(\mathsf{c}, D)$ assigns $D \in 2^{\mathcal{C}}$ to $\mathit{deps}(\mathsf{c})$. This requires the phase of c to be either start or pending. After this call, c reaches the commit phase. Let $\mathit{deps}^*(\mathsf{c})$ be the transitive closure of the *deps* relation starting from $\{\mathsf{c}\}$. Command c is stable once it is committed and so are its transitive dependencies in $\mathit{deps}^*(\mathsf{c})$.

Using these notions, we now define Byzantine leaderless state-machine replication (BLSMR). It guarantees the following properties:

Validity: If c is committed at a correct process then c is a command and all its dependencies are commands.

Agreement: For each command c, there exists $D \in 2^{\mathcal{C}}$ such that at a correct process p, if c is committed at p then $\mathit{deps}(\mathsf{c}) = D$.

Consistency: Consider that two correct processes p and q commit respectively two conflicting commands c and d with dependencies D and D'. Then, $\mathsf{d} \in D$ or $\mathsf{c} \in D'$.

Liveness: If a command is committed at some correct process then it is eventually committed at all correct processes.

By Validity, the graph is composed of state-machine commands. Agreement ensures that correct processes commit the same *deps* components. The third property enforces that the ordering of commands takes into account the conflict relation among them.

Example. Figure 1 depicts a run of BLSMR with four processes $\{p_1, \ldots, p_4\}$, and the commands $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ introduced earlier. In this run, p_1 submits command a, while p_2 submits c. Both processes also submit command b—this might happen when the system lags, as detailed in §4. The timeline in Figure 1 indicates the timing of these submissions. It also includes events during which the processes commit commands.

For some of these events, we detail the state of the dependency graph at the processes (right of Figure 1). In graphs G and H, command b is pending. All the processes eventually construct the same graph J. In J, commands a, b, and c are all committed. We have $\mathit{deps}(\mathsf{a}) = \{\mathsf{b}\}$ and $\mathit{deps}(\mathsf{b}) = \{\mathsf{a}\}$, with both $\mathit{deps}^*(\mathsf{a})$ and $\mathit{deps}^*(\mathsf{b})$ equal to $\{\mathsf{a}, \mathsf{b}\}$. The three commands a, b, and c are all stable in J.

3.2 Commands execution

Executing state-machine commands in BLSMR follows the same logic as Egalitarian Paxos [75]. In detail, processes execute commands once they are stable. This means that the command is committed, as well as its dependencies, and the dependencies of its dependencies, and so on. Execution takes place in the order encoded in the dependency graph, that is, the dependencies of a command execute before it. If a command is in a cycle (such as commands a and b in Figure 1), the full cycle executes in a batch. In this situation, commands execute according to some canonical order—e.g., using their identifiers.

To illustrate this mechanism, consider again Figure 1. Suppose that cyclic commands (a and b in the figure) get ordered in lexicographical order. A possible execution order for the three commands is abc. In this situation, a returns 0, the initial balance of Bob. As expected, the same results (final state and response values) are obtained with cab and acb.

Appendix A provides the full details of how the execution mechanism of BLSMR works. It also proves that BLSMR implements a replicated state machine, as defined in §2.

▷ Theorem 1. BLSMR implements a Byzantine fault-tolerant state-machine replication protocol.

In Figure 1, because c commutes with everything, it can be executed once committed. For instance, p_2 may execute it as soon as graph H is known. BLSMR can also expedite commands (return at commit) when it knows that the command is blind, i.e., there is no response value, and involved in no cycle. This happens in some applicative use cases. For instance, when the object under access has a single owner (e.g., asset transfer [50], per-tenant log) and anti-dependencies do

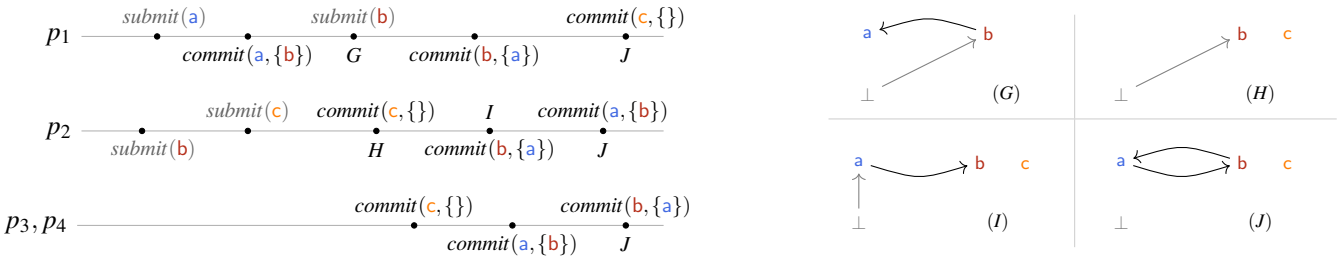


Figure 1: A run of BLSMR: (left) processes submit and commit commands a, b and c; (right) the dependency graphs formed at the four distributed processes.

not form due to reads [42]. In what follows, we focus on the general case and leave these (orthogonal) optimizations for future work.

4 An Abstract Machine

In this section, we introduce an abstract machine to implement BLSMR. The machine is defined as a composition of trusted services. We first explain how to embed trust in the values and variables a process manipulates, then we detail the machine.

4.1 Trust management

Trust is paramount in Byzantine fault-tolerant systems. It is implemented by verifying that the result of a computation is not tampered with by Byzantine processes. Below, we introduce an operator to lighten the description of algorithms that do such a verification.

Base notions. For starters, we model a stateless verification of trust. We say that a value x is *provable* if x embeds a proof, written $x.\Gamma \in \mathcal{P}$, where $\mathcal{P}$ is the universe of proofs. For some pair of values (x,y), with y provable, a function ϕ is *verifiable* if there exists a boolean function $check_\phi$, called *the verifier of* ϕ, such that $check_\phi(x,y)$ evaluates to true if and only if $\phi(x) = y$. Given a verifiable function ϕ, we next define a trusted assignment operator. This operator provides syntactic glue to check if two values are mapped by ϕ before doing the assignment. When this is not the case, an error is raised. More precisely,

$$var \xleftarrow{\phi,x} y \triangleq \textbf{if } check_\phi(x,y) \textbf{ then } var \leftarrow y \textbf{ else error}$$

To illustrate, consider a verifiable random function (VRF) [13,74]. Recall that given some pair of keys (SK, VK), where SK is a secret key and VK is the corresponding public verification key, a VRF provides jointly *(i)* an evaluator $Eval_{SK}(x)$ which given the key SK and some input x, returns an output y not distinguishable from a pseudo-random value, *(ii)* a prover $P_{SK}(x)$ that takes as parameter SK and outputs a proof π that y matches the input x and VK, and *(iii)* a verifier $V_{VK}(\pi,x,y)$, which verifies the proof. Using the previous definitions, $Eval_{SK}(x)$ and $V_{VK}(\pi,x,y)$ are a verifiable function and its verifier, respectively. The proof π generated by $P_{SK}(x)$ is attached to y to make it provable (i.e., $y.\Gamma = \pi$).

Trusted services. The notion of trust as defined above is stateless. We now extend it to apply to a stateful service as well. This is achieved by taking into account the history of the service. In detail, consider a service S and a history h. Following [28], we write $h|Correct$ for the projection of h to all correct processes. Relation $S(x) = y$ holds in h when for some history h' of S, with $h'|Correct = h|Correct$, S is invoked with input x and responds with y in h'. Service S is *verifiable* when there exists a boolean function $check_S(x,y)$, with y provable, such that $check_S(x,y)$ evaluates to true at a correct process iff $S(x) = y$ in the current history. Function $check_S(x,y)$ is called the verifier of S. By extension, the trusted assignment operator also applies to some verifiable service S:

$$var \xleftarrow{S,x} y \triangleq \textbf{if } check_S(x,y) \textbf{ then } var \leftarrow y \textbf{ else error}$$

Next, we turn our attention to the composition of services. We compose two services in a way that the second service verifies the output of the first one before doing any actual computation. More precisely, suppose some value x and a verifiable service T. A service S is *verifying wrt.* (T,x) when $S(y) = z$ for some (y,z) implies that $check_T(x,y)$ also holds. If S is also verifiable, it is said to be *trusted wrt.* (T,x).

Examples. Many practical systems abide by the above definitions. For instance, in a PKI architecture, or when enclaves are available [32], each process i is able to sign some computation $y = f(x)$ given an input x. Knowing x, a process may then verify that a result y was actually computed by i by applying an appropriate verifier $check_{f,i}(x,y)$.

Another example is (one-shot) PBFT [18, 25]. Function $check(x,y)$ returns *true* when there exists a view v and a set of $2f+1$ signed commit messages for y in v. PBFT can also verify that x is valid before it takes any step. In this case, denoting by $1_C(x)$ the verifiable function that indicates if an input is a command, PBFT is trusted wrt. $(1_C, _)$.

4.2 The machine

The abstract machine to implement BLSMR is depicted in Algorithm 1. It consists of two actions. An action executes once its preconditions (**pre:**) are all true. In what follows, we provide an overview of this construction and then detail its internals.

Overview. Algorithm 1 implements BLSMR with the help of two services: a service to discover dependencies (DDS), and a consensus service (CONS). The general idea of this construction is as follows: Upon submitting a command c, c is passed to DDS. The service returns commands conflicting with c; of any two conflicting commands, one sees the other. This response is then injected into consensus to ensure that the processes commit the same dependencies for c. In addition, Algorithm 1 also contains a fast path that skips consensus. This path is taken when the DDS service indicates so. Typically, the fast path is permitted by DDS when the command is conflict-free, that is, when there are no concurrent conflicting commands.

Internals. Besides the dependency graph, Algorithm 1 uses three variables. First, for each command c, $\mathsf{CONS_c}$ is an instance of consensus associated with c. As is common, this service offers a single operation *propose*() that returns the value upon which processes agree. The second variable is a dependency discovery service (DDS). This service has a single operation *announce*(c) that outputs a pair (D,b), with $D \in 2^{\mathcal{C}}$ and $b \in \{true, false\}$. Flag b indicates if the *fast path* that skips consensus can be taken (at line 11 in Algorithm 1). Furthermore, consider two pairs (D,b) and (D',b') such that $check_{\mathsf{DDS}}(\mathsf{c},(D,b))$ and $check_{\mathsf{DDS}}(\mathsf{c}',(D',b'))$ both evaluate to true. That is, (D,b) and (D',b') are valid outputs of announcing c and c′, possibly at faulty processes. The DDS service maintains the following invariants:

Validity If $\mathsf{d} \in D$, then d is a command.

Visibility If $\mathsf{c} \asymp \mathsf{c}'$, then $\mathsf{c} \in D'$ or $\mathsf{c}' \in D$.

Weak Agreement If $\mathsf{c} = \mathsf{c}'$ and $b = true$, then $D = D'$.

Liveness At a correct process, every invocation of *announce*(_) eventually returns.

The Validity property guarantees that the service only returns commands as dependencies. Visibility ensures that conflicting commands do not miss each other. This property is pairwise: D does not need to contain all the commands conflicting with c. For each pair of conflicting commands, one direction suffices, as this already fixes their execution order. Weak Agreement enforces that the dependencies announced in the fast path are returned everywhere else. That is, if a process p retrieves $(D, true)$ when announcing command c, any process that announces c must also retrieve D. Liveness guarantees that correct processes always make progress.

The third variable of Algorithm 1 is a weak failure detector ($\mathcal{FD}$) that indicates if a process is slow or not. Here, a process is slow when a command it coordinates does not commit in time. More precisely, it is required that: *(Completeness)* If a command c is never committed at a correct process p, then there is a time after which *coord*(c) is permanently suspected by p. This property adapts the strong completeness of [26] to Byzantine failures, as a faulty coordinator may stay alive yet never commit its command. A local timeout on uncommitted commands implements it, and, as accuracy is not required, this timeout can be arbitrarily short; in the extreme, suspecting all processes is valid. If the coordinator of some command c is suspected (line 7), the local process steps in to commit c; this is called a *recovery*.

Algorithm 1 BLSMR machine – code at process p

```
1:  Variables:
2:    deps                                          // dependency graph
3:    (CONS_c)_{c∈C}                                // one consensus instance per command
4:    DDS                                           // dependency discovery service
5:    FD                                            // weak failure detector
6:  submit(c) :=
7:    pre:  p = coord(c) ∨ (∃d, c ∈ deps(d) ∧ coord(c) ∈ FD)
8:          phase(c) = start
9:    eff:  deps(c) ← ⊥
10:         (D,b) ←DDS,c— DDS.announce(c)
11:         if b = false then D ←CONS_c,D— CONS_c.propose(D)
12:         send(c,D,b) to all
13: commit(c) :=
14:   pre:  recv(c,D,b)
15:         phase(c) ∈ {start, pending}
16:   eff:  deps(c) ←v,(c,D,b)— D
17:         send(c,D,b) to all                      // best-effort
```

Both the dependency discovery and consensus services are trusted: DDS is trusted wrt. $(1_{\mathcal{C}},_)$, while $\mathsf{CONS_c}$ is trusted wrt. $(\mathsf{DDS},\mathsf{c})$. Below, we detail the logic of Algorithm 1 and the rationale behind these two trust assumptions.

A process submits a command when it is the coordinator or performs a recovery (line 7). First, the command is announced to the system (line 10). The announcement generates a set of dependencies and a flag to indicate if the fast path is available. If not, the command goes through the *slow path* and consensus is called (line 11). Once a decision is taken, the process broadcasts it to everybody in the system (line 12). When this message is received, a verifiable function v is called (line 16), and the message is relayed to all on a best-effort basis, in case the coordinator is Byzantine (line 17). In practice, it can be implemented with pairwise anti-entropy, as CometBFT does for lagging nodes [30]. Function v takes as input the command (c), its dependencies (D), and the fast-path flag (b), and it returns D. The verifier is defined below:

$$check_v((\mathsf{c},D,b),D) \triangleq \textbf{if } b \textbf{ then } check_{\mathsf{DDS}}(\mathsf{c},(D,b)) \textbf{ else } check_{\mathsf{CONS_c}}(D,D)$$

Function v guarantees that the logic of the abstract machine is respected. In detail, if the fast path is taken, the pair (D,b)

must be the result of a call to the DDS service with input c. As DDS is trusted wrt. $(1_{\mathcal{C}}, _)$, c is necessarily a command. Alternatively, c has taken the slow path. In this case, the verifier checks that $\mathsf{CONS}_{\mathsf{c}}$ is called. If this holds, because $\mathsf{CONS}_{\mathsf{c}}$ is trusted wrt. $(\mathsf{DDS}, \mathsf{c})$, we also have the same guarantees as in the previous case: DDS is called with input c and returns D.

Example. To illustrate the BLSMR machine, let us consider again the run in Figure 1. Such a run may unfold as follows: After submitting command a, process p_1 retrieves $(\{\mathsf{b}\}, \mathit{false})$ from the DDS service. This indicates that a should go on the slow path. Process p_1 proposes $\{\mathsf{b}\}$ to $\mathsf{CONS}_{\mathsf{a}}$. This value is decided and p_1 commits a as illustrated in graph G. In Figure 1, p_2 knows about this decision at the end of the run, when its local graph is J. Unlike a, command c has no dependencies. Hence, c may have taken the fast path in Algorithm 1, skipping consensus. That is, p_2 retrieves $(\{\}, \mathit{true})$ from its call to $\mathsf{DDS}.\mathit{announce}(\mathsf{c})$.

In Figure 1, b is recovered by process p_1. Indeed, command b is proposed first by p_2 and later by p_1. Such a situation might occur as follows: Internally, p_2 retrieves $(\{\}, \mathit{false})$ when announcing b. Then, process p_2 is disconnected and detected as slow by p_1 ($p_2 \in \mathcal{FD}$ at p_1). Because b is a dependency of a, p_1 starts the recovery of b. It announces command b by calling the DDS service. This call retrieves $(\{\mathsf{a}\}, \mathit{false})$ and p_1 proposes $\{\mathsf{a}\}$ to $\mathsf{CONS}_{\mathsf{b}}$. As seen in Figure 1, this value is decided, leading to $\mathit{deps}(\mathsf{b}) = \{\mathsf{a}\}$.

Correctness. The theorem below establishes that Algorithm 1 is correct. The proof is in Appendix E.1. Two basic observations ensure that the construction is sound: First, the Visibility property of DDS guarantees that conflicting commands see each other, as required by the Consistency property of BLSMR. Second, decisions on the fast path are the same as proposals to consensus on the slow path. This comes from the Weak Agreement property of DDS. Hence, the BLSMR machine guarantees that processes eventually agree on the dependencies of each command.

▷ Theorem 2. Algorithm 1 implements BLSMR.

5 Efficient Solutions

Algorithm 1 explains *at a high level* how to implement BLSMR in a principled manner. It remains to detail efficient implementations of the DDS and CONS services. This is the purpose of the present section. For pedagogical reasons, we first present an EPaxos-like protocol with only a slow path. Then, we explain how to take the fast path when submitted commands are conflict-free (such as c in Figure 1). The three protocols in this section are *alternative* instantiations of the abstract machine: each one provides its own DDS and CONS services, and they are not composed with one another.

Algorithm 2 Acquiring dependencies à la EPaxos – code at process p

```
1:  Variables:
2:    BQS                                   // dissemination quorum system
3:    L                                     // log (local to p)
4:  announce(c) :=
5:    eff: send(c) to all
6:         wait until ∃Q ∈ BQS.quorums(),
               ∀j ∈ Q, ∃D_j ⊆ C, recv(c, D_j) from j
7:         D ← ⋃_{j∈Q} D_j
8:         D.Γ ← (j, D_j)_{j∈Q}
9:         return (D, false)
10: reply(c, q) :=
11:   pre: recv(c) from q
12:        1_C(c) = c
13:   eff: deps(c) ← L.conflicts(c)
14:        send(c, deps(c)) to q
15: check(c, (D, b)) :=
16:   let (j, D_j)_{j∈Q} ← D.Γ
17:   v ← ∧ check_BQS(Q)
18:       ∧ ∃q, ∀j ∈ Q, check_{reply,j}((c, q), D_j)
19:       ∧ D = ⋃_{j∈Q} D_j ⊆ C
20:       ∧ ¬ b
21:   return v
```

5.1 A first solution: à la EPaxos

In what follows, we implement BLSMR in the spirit of Egalitarian Paxos (EPaxos) [75]. For clarity, we consider a variation of EPaxos which does not use sequence numbers or a fast path.

Discovering dependencies. Algorithm 2 details the DDS service of EPaxos for the Byzantine case. At each process, the algorithm uses two variables. Variable L is a log. A call to $\mathit{append}(\mathsf{c})$ appends c to L, if it is not already there, and then returns the index of c in the log. Operation $\mathit{conflicts}(\mathsf{c})$ also appends c to L, then it returns all the conflicting commands before c in L. The second variable is a Byzantine quorum system. Byzantine quorum systems extend quorum systems to mask Byzantine failures [69]. Variable BQS encapsulates such a quorum system into a verifiable service that offers two operations: $\mathit{getQuorum}(v)$ returns a quorum for some input v, and $\mathit{quorums}()$ lists all the possible quorums. Its verifier takes as input a set of processes P, and returns *true* if and only if $P \in \mathit{quorums}()$.

To announce a command c, a process broadcasts it (line 5 in Algorithm 2). When some process p receives c, p verifies that c is a command (line 12). Then, it appends c to its log and computes the commands conflicting with it. This defines the dependencies of c according to p. Once computed, these dependencies are sent back to the announcer (line 14).

After receiving a set of dependencies from a quorum, the announcer calculates their union (line 7). Then, it generates and embeds the proof necessary to make the service verifiable (line 8). The proof contains enough evidence for other processes to verify that the announcement was legitimate. For

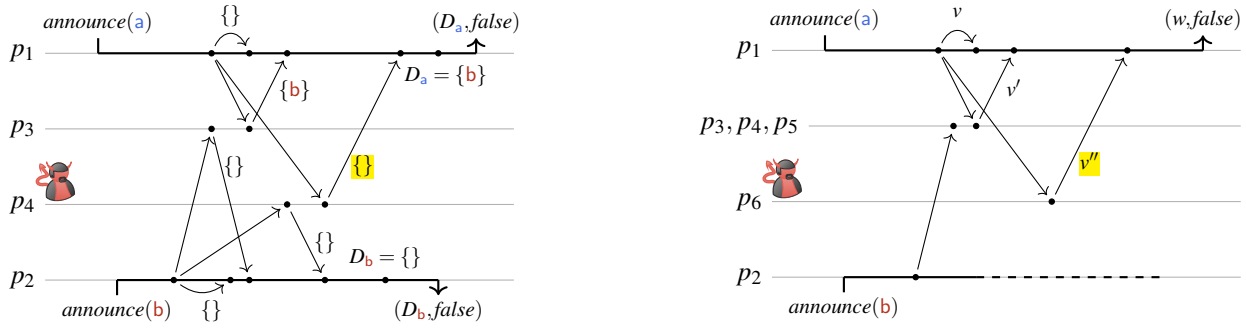


Figure 2: Fetching dependencies à la EPaxos (left) and Wintermute (right), $f = 1$.

each process in the quorum, the proof stores the dependencies it replied with. Once the proof is computed, the announcer returns the tuple $(D, false)$. The hardcoded value *false* indicates that there is no fast path in this implementation.

Function $check(\mathsf{c}, (D, b))$, defined in lines 15 to 21, is the verifier of Algorithm 2. It ensures that (D, b) is a valid output for c. For this, the function verifies the following facts: (line 17) a proper quorum was used; (line 18) for each member j of the quorum, the dependencies were actually computed by j; (line 19) D is the union of these dependencies and only contains commands; and (line 20) the fast path is not taken.

Variable *BQS* is a dissemination quorum system [69]. It ensures that any two quorums intersect over $f+1$ processes (the definition is recalled in Appendix B). As a consequence, at least one correct process reports all the conflicting commands it has seen so far. This ensures that Algorithm 2 guarantees the Visibility property of the DDS service. Building upon this observation, we prove that this construction is correct (Appendix E.2).

Proposition 1. *Algorithm 2 implements a trusted DDS service.*

Example. Figure 2(left) illustrates Algorithm 2. In this run, p_1 and p_2 concurrently announce two conflicting commands a and b. Quorum $Q = \{p_1, p_3, p_4\}$ replies to p_1, while b is announced using $Q' = \{p_2, p_3, p_4\}$. Process p_3 receives b before a, and replies $\{\mathsf{b}\}$ to p_1. The two commands are also received in this order at p_4. However, because p_4 is Byzantine, it decides to hide this information and replies $\{\}$. Nonetheless, because Q and Q' intersect over $f+1 = 2$ processes, this is not a problem. By taking the union over all replies, p_1 computes $(D_\mathsf{a} = \{\mathsf{b}\}, false)$ as the result of announcing a. It follows that a sees the conflict with b, as required by Visibility.

Agreement. To build a full protocol using the BLSMR machine (see §4), we also need a trusted consensus service. Any BFT consensus protocol is usable in this task. For instance, we can use (single-shot) HotStuff [18] or PBFT [25], asking that processes participate in CONS_c only if $check(\mathsf{c}, (D, b))$ holds for the verifier in Algorithm 2. To make consensus verifiable, we then define a verifier $check_{\mathsf{CONS}}$. With PBFT, this function is implemented by using the $2f+1$ signed commit messages of the decision: these messages form the proof, and function $check_{\mathsf{CONS}}$ verifies that they match. The consensus service runs over all processes in the system (which are by assumption $n \geq 3f+1$). As indicated in §4.2, one instance (CONS_c) exists per command c. For performance, the leader of the first view in CONS_c is set to $coord(\mathsf{c})$. In synchronous failure-free runs, PBFT then permits committing command c at $coord(\mathsf{c})$ after five delays: two message delays for the announcement of the command, followed by three (pre-prepare, prepare, and commit) for the commitment of its dependencies.

5.2 Second solution: Wintermute

At a process, dependencies accumulate over time in Algorithm 2. To bound metadata, it is necessary to cut the dependency graph periodically, e.g., using checkpoints [62, 93]. Our next solution solves this concern efficiently and adds a fast path.

5.2.1 Managing metadata

To reduce metadata usage in BLSMR, we need to compact dependencies. We address this point and then explain how to prevent Byzantine processes from interfering in the protocol.

Compaction. When a command c is created, it gets assigned a sequence number, written $\mathsf{c}.sn$. This sequence number is generated by the client, similarly to a nonce in Ethereum [20]. Dependencies are encoded (over-approximated) as vectors of sequence numbers. For instance, assume three clients $\{x, y, z\}$, using Algorithm 2 for reference. At line 13, a compact representation of $deps(\mathsf{c})$ could be the following: $deps(\mathsf{c}) = \{x \mapsto 2, y \mapsto 7, z \mapsto 4\}$. Dimension x of this vector indicates that the first two commands of client x are dependencies of c.

Compaction is a metadata optimization only. Since sequence numbers are assigned by the clients, it adds no message delay to the protocol. Dependencies may be compacted further, as vectors of size $O(n)$ indexed by processes, by

Algorithm 3 Acquiring dependencies in Wintermute – code at process p

```
1:  Variables:
2:    BQS                                   // witnessing quorum system
3:    L
4:  announce(c) :=
5:    eff: send(c) to all
6:         wait until ∃Q ∈ BQS.quorums(),   // largest if multiple choices
               ∀j ∈ Q, ∃D_j ⊆ C, recv(c, D_j) from j
7:         D ← ⌊⊔⌋_{j∈Q} D_j
8:         D.Γ ← (j, D_j)_{j∈Q}
9:         return (D, fast(c, D))
10: reply(c, q) :=
11:   pre: recv(c) from q
12:        1_C(c) = c
13:   eff: deps(c) ← L.conflicts(c)
14:        send(c, deps(c)) to q
15: check(c, (D, b)) :=
16:   let (j, D_j)_{j∈Q} = D.Γ
17:   v ← ∧ check_{BQS}(Q)
18:       ∧ ∃q, ∀j ∈ Q, check_{reply,j}((c, q), D_j)
19:       ∧ D = ⌊⊔⌋_{j∈Q} D_j
20:       ∧ b = fast(c, D)
21:   return v
22: fast(c, D) :=
23:   let (j, D_j)_{j∈Q} = D.Γ
24:   b ← ∧ Q = ℙ
25:       ∧ ∀j ∈ Q, D_j = D
26:   return b
```

delegating the assignment of sequence numbers to the coordinators. This requires a verifiable sequencing service, which runs in parallel with the announcement of the command and is thus off the critical path. This optional optimization is detailed in Appendix D. For readability, the examples below index vectors by processes, as if this optimization were in use.

Compact dependencies reduce metadata. However, they also open a new angle of attack: when queried for dependencies, a Byzantine process can report illegitimate (fake) sequence numbers. Below, we illustrate this concern and explain how to cope with it.

Threshold union. A base observation is that a dependency is for sure legitimate when $f+1$ processes report it. Hence, to evade the attack, we discard dependencies that appear at most f times. To achieve this, we use the threshold union operator [43]. In detail, consider that Q is a quorum and let $(D_j)_{j\in Q}$ be dependencies reported by the processes in Q. The threshold union operator is defined as follows: $\lfloor\sqcup\rfloor_{j\in Q} D_j = \bigcup\{\mathsf{c} : count(\mathsf{c}) \geq f+1\}$, where $count(\mathsf{c}) = |\{j \in Q : \mathsf{c} \in D_j\}|$. In other words, the operator returns all the commands reported at least $f+1$ times in the quorum (see Appendix C).

To illustrate, consider the run in Figure 2(right). In this figure, a and b are submitted by p_1 and p_2, respectively, with $\mathsf{a}.sn = 2$ and $\mathsf{b}.sn = 3$. Command b is received before a at the processes $\{p_3, p_4, p_5\}$. These processes report vector v', while p_1 and p_6 report v and v'', respectively. Suppose $v = \{p_1 \mapsto 1, p_2 \mapsto 2\}$, $v' = \{p_1 \mapsto 1, p_2 \mapsto 3\}$, and $v'' = \{p_2 \mapsto 99\}$. If p_2 has not submitted more than 3 commands yet, a Byzantine node—namely p_6 in Figure 2(right)—tries to inject illegitimate dependencies with v''. Applying $\lfloor\sqcup\rfloor_{j\in Q} D_j$, with $f = 1$ and $Q = \{p_1, p_3, \ldots, p_6\}$, p_1 computes the dependencies $w = \{p_1 \mapsto 1, p_2 \mapsto 3\}$. Hence, the threshold union operator prevents p_6's attack.

The threshold union operator ensures that only submitted commands are reported. However, it might also remove legitimate dependencies. For instance, in the above example, $p_2 \mapsto 4$ is removed but it could be legitimate, provided p_2 had already submitted enough commands. Removing a legitimate dependency is not a problem in itself. Recall that Visibility is a pairwise property (§4): for two conflicting commands, it suffices that one of them is a dependency of the other, and this already fixes their execution order. To ensure that the threshold union keeps at least one of the two dependencies, we use quorum intersections of size $3f+1$. We call such a quorum system a *witnessing quorum system* (WQS). The formal definition is given in Appendix B.2.

To illustrate why we need $3f+1$ processes in the intersection, let us go back to Figure 2(right). Assume that a and b are announced using quorums $Q = \{p_1, p_3, \ldots, p_6\}$ and $Q' = \{p_2, \ldots, p_6\}$, respectively. The intersection $Q \cap Q' = \{p_3, \ldots, p_6\}$ contains four processes. Among them, one can be faulty and may not report anything useful. At $f+1 = 2$ processes, either command b is received before a, or the converse holds. Thus, one of the two commands is reported by enough processes, ensuring the Visibility property of the DDS service. In Figure 2(right), the first case happens, since processes $\{p_3, p_4, p_5\}$ reply v' to p_1. Vector $w = \lfloor\sqcup\rfloor_{j\in Q} D_j$ encodes that b is a dependency of a.

5.2.2 Algorithm

Building upon the above techniques, we now present another implementation of the BLSMR machine. This new protocol is called *Wintermute*.

Discovering dependencies. The DDS service of Wintermute appears in Algorithm 3. The differences with Algorithm 2 are highlighted in dark blue. Commands have sequence numbers generated by either clients or a sequencing service, allowing dependencies to be compacted. Algorithm 3 uses a witnessing quorum system instead of a dissemination one (line 2). The threshold union operator trims illegitimate dependencies (lines 7 and 19).

Fast Path. A key difference with Algorithm 2 is the fast path at line 9. The fast path is permitted when function $fast()$ returns *true*. This happens provided that all the processes answer (line 24) and that they report the exact same dependencies (line 25). To give all the processes a chance to answer, the announcer uses the largest quorum available when the wait at line 6 ends. Since quorums are closed under supersets (Ap-

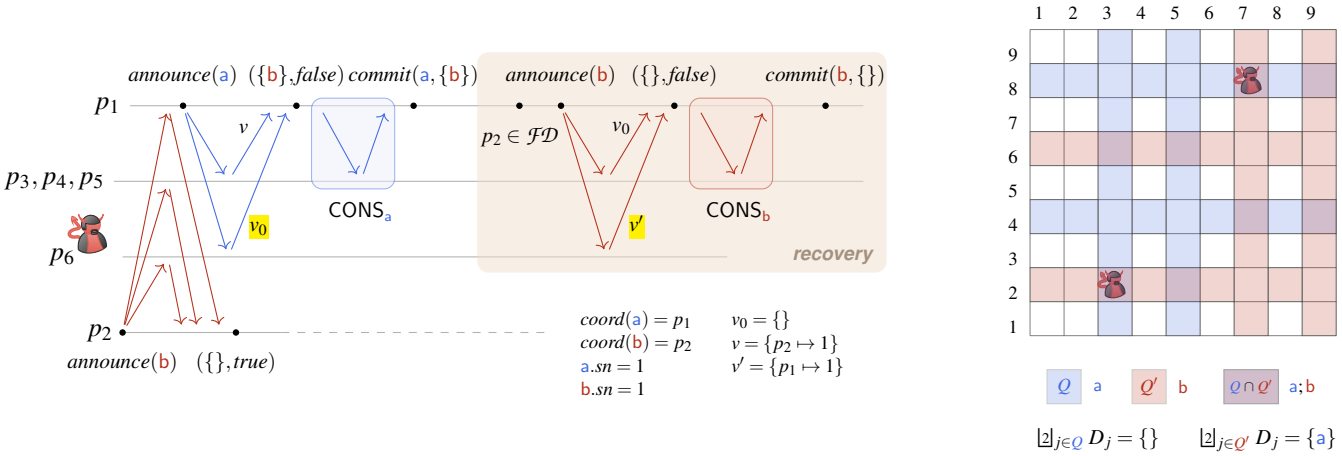


Figure 3: Illustration of the BLSMR protocols: (left) Wintermute with $n = 6$, $f = 1$; (right) 3Jane uses a grid-like quorum system with $n = 81$, $f = 2$.

pendix B.1), this quorum consists of all the processes that replied at that time. In a failure-free synchronous run, all the replies arrive together, and thus this quorum contains all the processes. In practice, the announcer may wait a little longer, e.g., using a timeout, to collect more replies.

The intuition behind the fast path is as follows: Consider that p retrieves $(D, true)$, while q obtains $(D', _)$ when announcing command c. Because p takes the fast path, every process reports $\mathsf{d} \in D$ to p. A correct process always sends the same reply for c (Lemma 21), and the quorum used by q contains at least $2f+1$ correct processes. Hence, d is reported at least $f+1$ times to q, and it is necessarily included in D'. Conversely, if $\mathsf{d} \in D'$, then d is reported $f+1$ times to q. It follows that at least one correct process that reports to q sees d. This process must answer to p for the fast path to be permitted. Then, because all the processes must report the same dependencies to p, d is in D.

Using the above line of arguments, we can establish that Algorithm 3 satisfies Weak Agreement. Proving the other properties of the DDS service is similar to our first solution. The details are in Appendix E.3.

Proposition 2. *Algorithm 3 implements a trusted DDS service.*

Consensus. Since Wintermute implements the BLSMR machine, it may use any consensus protocol for the CONS service. We notice that in the common case only the coordinator of command c proposes a value to CONS_c. Consequently, a two-step algorithm, that is, an algorithm solving consensus in a single round-trip when the leader is stable, is the most appropriate. *Quorum* [6] returns after a single round-trip in which it contacts $3f+1$ processes. A more general solution is the one proposed in [60]. This protocol uses $3f+2e-1$ processes, offers the same latency as *Quorum*, and resists in the fast path to up to $e \geq 1$ Byzantine processes. Either protocol can be used in Wintermute; our simulations (§6) use *Quorum*.

Byzantine dependencies. We summarize how Wintermute copes with Byzantine processes during dependency discovery. First, a Byzantine process may omit dependencies in its reply. This is masked by quorum intersections: any two quorums share at least $2f+1$ correct processes, so Visibility holds. Second, a Byzantine process may report fake dependencies, e.g., a sequence number that was never issued. Such dependencies are discarded by the threshold union, as any dependency in D is reported by at least one correct process (Validity). Third, a Byzantine coordinator cannot forge the dependencies or the fast-path flag. Indeed, the verifier recomputes both of them from the signed replies stored in the proof (lines 16 to 20), and it is checked by every process before committing (line 16 of Algorithm 1). In the EPaxos-like protocol (Algorithm 2), dependencies are combined with a plain union, so a Byzantine process may add commands to D. However, these are actual commands (Validity), and an additional dependency is harmless: it only delays execution until that command commits, which recovery guarantees.

End-to-end execution and performance. We now summarize how a command c submitted at $p = coord(\mathsf{c})$ flows through Wintermute. We illustrate this with the run of Figure 3(left), where p_1 and p_2 submit the conflicting commands a and b, respectively. Process p broadcasts c (line 10 in Algorithm 1, then line 5 in Algorithm 3). Every process that receives c appends it to its log and replies with the conflicting commands that precede c (lines 10 to 14). Process p then computes the threshold union D of the replies of the largest available quorum, together with its proof (lines 6 to 8).

- *Fast path (2 delays).* If all the processes report the same dependencies, p skips consensus and broadcasts $(\mathsf{c}, D, true)$ (line 12 in Algorithm 1). In Figure 3(left), no process reports a dependency for b, so p_2 takes the fast path.

- *Slow path (4 delays).* Otherwise, p proposes D to $\mathsf{CONS_c}$, in which it leads the first view (line 11). With the two-step consensus above, this adds a single round-trip. This is the case of a: p_2 is slow and does not reply in time, and the Byzantine process p_6 omits b (v_0). Process p_1 proposes the threshold union over $Q = \{p_1, p_3, \ldots, p_6\}$, that is v, which encodes $\{\mathsf{b}\}$, and this value is decided.
- *Recovery.* If p is suspected by a process q that knows a command depending on c, then q executes the above steps for c (line 7). If p took the fast path with D, then q obtains D as well (Weak Agreement); otherwise, consensus ensures that they agree. In the figure, p_2 is slow (dashed line), and its commit message has not reached the other processes when p_1 recovers b. The threshold union discards a, which p_6 maliciously reports (v'), and p_1 obtains $\{\}$, as p_2 on the fast path.

In a failure-free synchronous run, the commands conflicting with a conflict-free command are already executed everywhere when it is submitted. Hence, all the processes report the same dependencies, and the command executes at its coordinator after a single round-trip (see §3.2). This means that the protocol is fast, as stated in the result below (the proof is in Appendix F). This is a well-known lower bound [63]. To the best of our knowledge Wintermute is the first protocol to reach it.

▷ Theorem 3. Wintermute is a fast Byzantine fault-tolerant state-machine replication protocol.

5.3 Third solution: 3Jane

Blockchains span hundreds to thousands of machines. To operate at this scale ($f \ll n$), our third protocol, 3Jane, avoids involving all processes in ordering every command.

Improving scalability. We introduce several modifications to the design of Wintermute that help to scale the approach. These are detailed below.

So far, we have considered that DDS and CONS run over all the machines. *This is not necessary.* A first refinement is to deploy DDS everywhere but execute CONS only at a subset of machines (e.g., just $3f+1$).

The following observation motivates a second refinement: in BLSMR, $\mathsf{CONS_c}$ and $\mathsf{CONS_d}$ *may run in parallel*. We leverage this to lower the load in the manner of Algorand [46]: for some command c, the processes running $\mathsf{CONS_c}$ are chosen using a verifiable random function that takes as input c.

In the fast path, Wintermute contacts all the processes. Hence, its load, that is, the worst-case probability a process gets chosen to order a command, is 1. Instead, 3Jane contacts a single quorum picked uniformly at random from a grid-like witnessing quorum system (Appendix B.2). If this quorum does not reply in time, the announcer picks another one; by Availability, it eventually reaches a correct quorum.

The above modifications drop the fast path but reduce the load of the protocol. In particular, when $f \ll n$, each process only orders a small fraction of the state-machine commands. Such a result is stated in the proposition below. The corresponding proof appears in Appendix F.

Proposition 3. *The load of 3Jane is* $O(\sqrt{f/n})$.

When the Byzantine adversary is weak ($f \ll n$), the load of 3Jane satisfies that: $lim_{n\to\infty}\ \mathcal{D} = 0$. This comes from the fact that both the quorum of the DDS service and the $3f+1$ processes running consensus are small and change from one command to another. In such a context, the protocol scales the cost of ordering the state-machine commands with the number of processes.

The changes introduced in 3Jane do not impact the overall logic of our approach. Hence, the correctness of this protocol follows from Proposition 2 and Theorem 2.

Example. 3Jane is illustrated in Figure 3(right). In this figure, each quorum consists of two rows plus two columns. This tolerates a small number of failures, i.e., at most $f = 2$ when $n = 81$, but better spreads the load: each process has around a 39% chance to participate in the announcement of a command, compared to $\sim 52\%$ with the (usual) dissemination quorum system.

In Figure 3(right), command a is announced with quorum Q, while b uses Q'. Processes at the intersection receive command a first, then b. In the figure, p_{ij} denotes the process at row i and column j. There are two Byzantine processes, p_{23} and p_{87}. Since $f = 2$, whatever these two processes report, $\lfloor 2 \rfloor_{j \in Q} D_j = \{\}$. On the other hand, command b is announced with dependencies $\{\mathsf{a}\}$. This comes from the fact that $\lfloor 2 \rfloor_{j \in Q'} D_j = \{\mathsf{a}\}$. For instance, the correct processes $\{p_{63}, p_{65}, p_{25}, p_{47}, p_{49}\}$ all report a.

5.4 Efficiency

Table 1 compares the proposed solutions against state-of-the-art approaches. It mentions when a protocol is *leaderless*, that is, when the protocol offers the same latency everywhere. In particular, protocols that rotate their leader for performance (e.g., Chained HotStuff, HotStuff-1, Tendermint) are considered leaderless, whereas protocols that change their leader only to restore liveness (e.g., PBFT and vanilla HotStuff) are not. The table also indicates protocols that leverage *commutativity*, that is, protocols that order only non-commuting (conflicting) state-machine commands. In case only commuting commands are submitted, such protocols boil down to reliable broadcast [62, 77].

The comparison of the protocols in Table 1 focuses on the common case, namely, when the system has no failures and behaves synchronously. Each client is local to some process replicating the state machine (proxy model). Measures are taken in the critical path, from the moment a command is submitted to when it is executed at a replica process.

Table 1 lists the following metrics of interest:

- $\mathcal{L}$ (conflict-free latency): Worst-case number of message delays in the critical path for a conflict-free command during a failure-free synchronous run (see §2.2). Conflict-free commands are defined in Definition 2.
- $\mathcal{M}$ (message complexity): Total number of messages exchanged in the critical path.
- $\mathcal{A}$ (authenticator complexity): Total number of signature verifications in the critical path. This includes the verification of the dependencies by the participants of consensus.
- $\mathcal{D}$ (load): Worst-case probability that a process is part of the critical path, as for quorum systems [70].
- $\mathcal{R}$ (resilience): Maximum number of failures a protocol tolerates.

Wintermute permits executing a conflict-free command after a single round-trip. It is fast in the sense defined in §2.2. To the best of our knowledge, Wintermute is the first BFT SMR protocol to achieve this property.

6 Evaluation

In this section, we evaluate our approach using a discrete-event simulator and compare it against state-of-the-art solutions. We answer the following research questions: *(RQ1) How does the latency of Wintermute compare with the state of the art in simulated geo-distributed deployments, under a realistic payment workload?* §6.3.1 *(RQ2) How does the fast-path ratio vary when conflicts increase?* §6.3.2 *(RQ3) How do Wintermute and 3Jane scale with the number of replicas?* §6.3.3

6.1 Simulator

Our evaluation campaign runs on a deterministic framework written in Rust and called DScale [83]. The framework simulates a distributed network as a set of processes that exchange messages using shared memory. DScale is a discrete event-based engine. DScale uses one centralized clock to time the reception and sending of messages. It advances time only when messages are exchanged, and all local computations are considered free and instantaneous. All the times reported in this section are expressed in milliseconds.

The simulations are deterministic and parametrized by a seed given at the start. The simulator provides two scheduling modes: sequential and parallel. To better accommodate the nature of the protocols and scale, our evaluation campaign uses the parallel mode. In this mode, simulations are distributed across an MPI cluster to be computed in parallel. Each simulation instance uses all the cores available on the machine where it runs.

6.2 Benchmark Settings

Baselines. As in recent works [7, 47], we compare our protocols against HotStuff and Bullshark. These two protocols are standard representatives of the two main families of BFT SMR protocols, namely leader-based and DAG-based protocols. Variants of both have been deployed in production blockchains (e.g., Diem and Sui, respectively). More recent protocols, such as HotStuff-1 [55] and Mysticeti [7], lower this latency. Table 1 compares them analytically with our protocols: they need at least three message delays to commit a command, whereas Wintermute needs two on its fast path. Moreover, they leverage the commutativity of commands only partially (Mysticeti) or not at all.

We compare the BFT SMR protocols listed below. Because the evaluation campaign focuses on the steady state, only the failure-free path in each protocol is implemented; in particular, the recovery and the relay of decisions (lines 7 and 17 in Algorithm 1) are not. Cryptographic operations and real transaction execution are omitted.

- HotStuff is a leader-based solution [100] that achieves linear message complexity. The protocol employs a three-phase commit protocol and a cryptographic chaining of quorum certificates to validate operations. HotStuff rotates the role of leader among the replicas.
- Bullshark [89] is a DAG-based protocol for partially synchronous systems with a fast-path optimization.
- Wintermute implements the abstract BLSMR machine using a two-step consensus protocol and allowing conflict-free commands to take a fast path. For consensus, we use Quorum [6], which commits after a single round trip in the absence of faults. The wait at line 6 of Algorithm 3 is implemented with a timeout of 500 ms: a command takes the fast path if all the replicas reply before the timeout with the same dependencies, and the slow path otherwise.
- 3Jane lowers the system load at the cost of forcing all commands to go through (Quorum) consensus. In all experiments, f is set to $\lfloor(\sqrt{n}-2)/3\rfloor$, i.e., the highest number of failures tolerated by the witnessing BQS.

BLSMR is implemented in $1,037$ SLOC, while Bullshark and HotStuff span 498 SLOC and 153 SLOC, respectively. Further details about the codebase and scripts are available in the "Artifacts" section of the paper.

Setup. The evaluation campaign runs on an MPI cluster of 19 machines. Each machine is equipped with an Intel i7-13700K processor with 24 hyperthreaded cores and 128 GB of RAM. We consider a varying number of replicas (n) and fault-tolerance threshold (f). Replicas are pair-wise connected. We evaluate two topologies: *(i)* a uniform topology, where the distance between any two replicas is constant and set to 100 ms, and *(ii)* a terrestrial topology simulating a planet-scale system. In this topology, replicas are scattered across datacenters, each located in a region. In total, we use eight regions: us-east-

1 (Washington), us-west-2 (Vancouver), ca-central-1 (Montreal), eu-west-1 (Dublin), eu-central-1 (Frankfurt am Main), ap-northeast-1 (Tokyo), ap-southeast-1 (Singapore), and ap-southeast-2 (Sydney). Except in the scalability experiment (see §6.3.3), the network delay between two regions follows a symmetric normal distribution. The mean of the distribution is derived from the AWS All-Regions Ping dataset [81], and its standard deviation is 0.593 ms, half the round-trip standard deviation in the CALM Virginia–Frankfurt sample [53, 54] (assuming perfectly correlated directions). Unless stated otherwise, each simulation has a time budget of 2,000 s.

Workloads. We use two workloads. The latency experiment (§6.3.1) uses a payment workload, while the conflict-rate and scalability experiments (§6.3.2 and §6.3.3) use a storage workload. Clients are virtual and run on the replicas.

The storage workload represents a dummy storage service [49]. Clients issue a single-key command chosen at random. The distribution of these accesses can be either uniform (default) or zipfian. The zipfian coefficient is 0.99, as in YCSB [31]. Each client submits commands at regular intervals; the interval varies in the conflict-rate experiment and is 500 ms in the scalability experiment.

The payment workload follows the synthetic VISA workload [37]: it replays 540,000 one-unit transfers among 10,000 accounts over 300 seconds, at an aggregate rate of 1,800 transactions per second. Each replica hosts ten virtual clients, and transfers are assigned to these clients in round-robin order. The same seeded sequence of sender and receiver pairs is used for every protocol and topology; two transfers conflict if they share an account.

6.3 Results

6.3.1 Latency

In this experiment, clients run the VISA payment workload (§6.2). We fix $f = 1$ for all the protocols. Since a single failure is tolerated, HotStuff and Bullshark use $n = 4$ replicas, while Wintermute and 3Jane use $n = 6$ and $n = 25$, respectively. The simulation runs for the duration of the trace (300 seconds), and the CDF includes only the commands that finish before then: Wintermute and 3Jane measure submission to stability, while HotStuff and Bullshark measure submission to first observed commit or DAG ordering.

Figure 4 presents the experimental results for the two topologies. In the uniform topology, HotStuff has a latency of 600–900 ms, with a median of 775 ms. The lower end corresponds to its three phases, while the additional delay comes from reaching the leader and being included in a decision as the leader rotates.

Bullshark first disseminates a command to the replicas, which include it in their next DAG vertex. Each DAG round is a consistent broadcast [21] and lasts three message delays. A command included in an even round can be ordered after

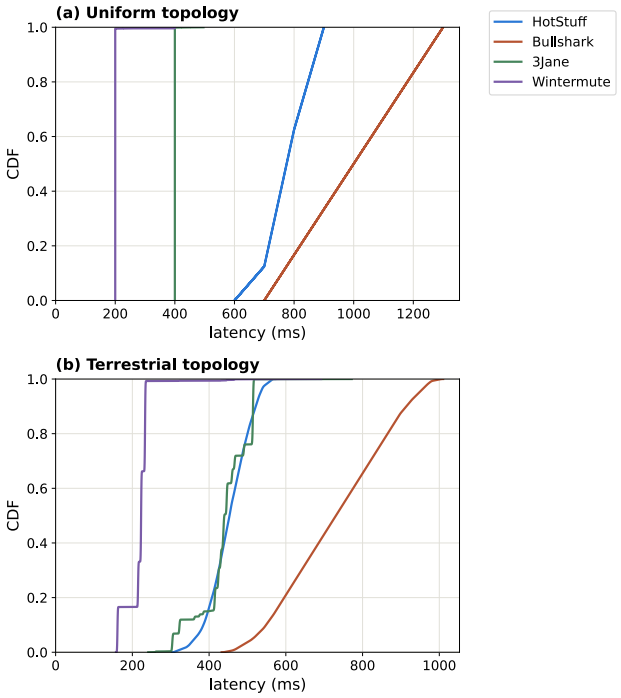


Figure 4: Cumulative distribution function (CDF) of sampled command latency under the uniform (a) and terrestrial (b) network topologies.

the next even-round leader observes $f + 1$ votes for its anchor, giving a latency of 700–1,000 ms. Commands included in an odd round wait for the next anchor, adding another round of up to 300 ms. The resulting latency ranges from 700 to 1,300 ms, with a median of 1,000 ms.

Wintermute achieves a median latency of 200 ms: 99.61% of commands take the fast path, while 0.39% take the slow path. Only a small fraction of the commands suffers a chaining effect, in which a command waits for its dependencies to commit. 3Jane has a median latency of 400 ms.

In the terrestrial topology, Wintermute takes the fast path for 99.3% of commands and the slow path for 0.7%; 47 commands (0.01%) experience a chaining effect. Its median latency is 222.79 ms, compared with 439.75 ms for 3Jane, 453.21 ms for HotStuff, and 730.56 ms for Bullshark. Thus, Wintermute has approximately half the median latency of HotStuff in this topology.

6.3.2 Impact of the conflict rate

For Wintermute, we also evaluate the execution latency when the conflict rate changes in the terrestrial topology. We use the dummy storage workload with 32 conflicting keys, and vary the submission interval to obtain different conflict rates.

The results are reported in Figure 5. In this figure, the first y-axis indicates the average execution latency and the second

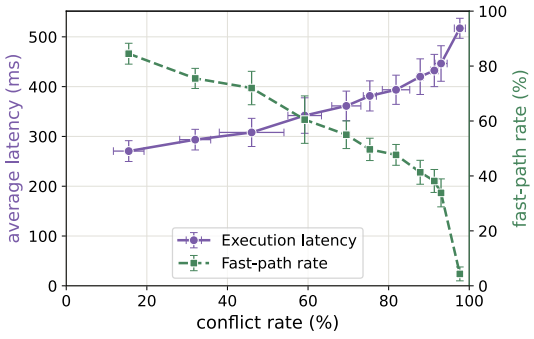


Figure 5: Execution latency in Wintermute under varying conflict rates.

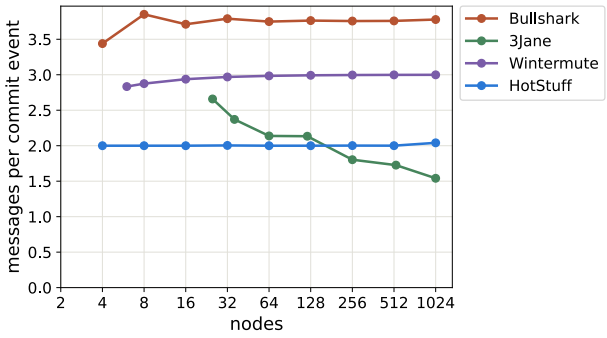


Figure 6: Load as the system scales up.

one the fast-path ratio. The x-axis is the observed conflict rate. Each submission interval is evaluated several times to obtain error bounds (standard deviation) on the results. These are reported as vertical and horizontal intervals in Figure 5.

The average latency of Wintermute rises from approximately 271 ms at a 16% conflict rate to 517 ms at a 98% conflict rate. The fast-path ratio falls from approximately 85% to 4% over the same range. The simulator models a perfect system without implementation bottlenecks that would lead to artifacts in the experiments. As a consequence, it is expected that the (average) execution latency follows an inverse trend to the fast-path ratio. This is what one observes in Figure 5. In this experiment, the Pearson's correlation coefficient between these two random variables is -0.9938.

6.3.3 Scalability

We now evaluate the scalability of each competitor. For this, we use the storage workload [49] and scale the terrestrial topology to include up to 1,024 replicas. This is on the order of the size of existing blockchain networks, such as Solana [87] and Internet Computer [36], but smaller than Bitcoin [14] and the recent Beacon Chain in Ethereum [12].

To simplify the simulation at large scale, we consider that any two regions are connected by a link whose latency is constant and equal to the speed-of-light propagation time between them. The storage service holds 10M keys. Clients access it using a zipfian distribution, sending a command every 500 ms.

The results of the experiment are reported in Figure 6. In this figure, the y-axis depicts the load of each protocol. It is measured as the *worst-case number of processed messages per commit event at a replica*. This empirically reflects how often a replica is picked to order a command—see our definition of load in §5.4.

Figure 6 shows that the load of HotStuff, Bullshark and Wintermute remains roughly constant when the system scales up. This is expected: in all of them, the work of ordering commands is spread over the replicas (through leader rotation in HotStuff and Bullshark).

With Wintermute, the fast path requires a follower replica to process two messages per commit event (the initial announcement and the commit broadcast). Taking into account the scatter-gather replies handled by the coordinator, the message load per replica scales as $(3n-1)/n = 3-1/n$. As shown in Figure 6, this load grows smoothly from 2.83 with 6 replicas to 3.0 at 1,024 replicas.

	Bullshark	HotStuff	3Jane	Wintermute
32 (36)	383 ± 71	251 ± 5	201 ± 29	103 ± 14
1,024	384 ± 71	251 ± 1	211 ± 43	103 ± 14

Table 2: Latency (ms) at different scales, with constant speed-of-light delays between regions.

Committing an anchor in Bullshark requires two DAG rounds, each implemented via consistent broadcast [21]. Because committing an anchor orders its entire uncommitted causal history at once, the message overhead of these consistent broadcast rounds is amortized across all payload commands. As a result, Bullshark maintains a stable load per commit event as the system scales. 3Jane is the only protocol whose load decreases with additional replicas; from 256 replicas on, it achieves the best load of all competitors. This is expected: when the system is larger, the fraction of replicas contacted in the critical path gets smaller.

Table 2 reports the average latency during the experiment at two different scales: 32 and 1,024 replicas. Because inter-region delays are lower and constant in this experiment, these latencies are lower than in Figure 4 and should not be compared with them. Notice that for 3Jane, we use 36 replicas (the closest perfect square) instead of 32 to build the grid quorum system.

Interestingly, the latency does not change much as the system scales up. This stems from the fact that replicas are distributed across a fixed set of geographic regions. Such a setting also explains why the standard deviation for HotStuff decreases, from 5 ms with 32 replicas down to 1 ms with

1,024 replicas. Conversely, for 3Jane, the standard deviation increases up to 43 ms due to the geographical heterogeneity of the grid quorums. Nonetheless, 3Jane achieves the second-best latency among all competitors in this experiment, right behind Wintermute.

7 Related work

Protocols. PBFT [25] provides the first practical BFT SMR solution for partially-synchronous systems. This solution is similar to Paxos [61] for the crash-failure case, requiring additional message exchanges to verify the computation performed at the leader. This additional work relates to the Dolev–Reischuk lower bound [39], which shows that Byzantine agreement requires $\Omega(n^2)$ messages in the worst case.

Zyzzyva [58] introduces a fast path in BFT SMR called speculative execution. When the fast path fails, it is necessary to go through a slow path. The slow path of Zyzzyva was found flawed and later corrected in [2]. Several works [1,6,48] also propose to expedite the execution of commands. In particular, Quorum [6] shortens this to just two message delays, which is optimal. To reach this lower bound, a client sends its command to all the replicas that execute it optimistically. If all replicas return the same response and their command histories are identical, the client returns right away. Otherwise, the client generates an abort history and the command goes through a slow path implemented with PBFT. As a consequence, the conflict-free latency of Quorum is 5, as reported in Table 1.

In [56], the authors propose a BFT SMR protocol called DAG-Rider. To avoid equivocation, this algorithm is built on top of a reliable BFT broadcast protocol (e.g., [17, 23]). A run is split into rounds, during which each process broadcasts a single proposal. A process may move to round $r+1$ once it has heard from $2f+1$ proposals in round r. When it moves to round $r+1$, the process chains its proposal to the $2f+1$ proposals in round r. Once enough rounds have passed, DAG-Rider decides locally by just looking at the DAG of the execution and querying a random oracle.[1] The above technique is extended and made practical in [35]. Bullshark [89] adds a fast-path optimization and further reduces costs. The authors report that, during synchrony periods, the protocol commits in two DAG rounds, that is, six message delays. However, this count starts once the command is included in a vertex of the DAG. A replica includes a command only in its next vertex, which may require waiting for a full round (three message delays). Hence, the conflict-free latency of Bullshark is 9, as reported in Table 1. As observed in §6, a command included in an odd round may even wait for one more round. Mysticeti [7] is a recent DAG-based protocol that permits expediting commands that never conflict.

The use of commutativity in SMR is introduced in [62, 77]. PGB [79] adapts Generic Broadcast [77] to Byzantine failures. When a single pair of conflicting commands is detected, the system runs a global consensus. Byblos [10] is based on the non-skipping timestamps technique introduced in [11]. It requires $n \geq 4f+1$ processes and has $O(n^2)$ message complexity. The protocol only leverages commutativity at execution time. Moreover, ordering requires a slow path that necessitates always running an agreement protocol.

A related line of work shows that Byzantine consensus can be avoided altogether for a restricted class of commands. The work in [50] proves that asset transfer, the operation underlying a cryptocurrency, has consensus number one: it can be implemented wait-free using reliable broadcast alone, because a double-spend is detected locally as an equivocation on the sender's account. FastPay [9] and Astro [29] turn this observation into practical protocols. Sui Lutris [15], and more recently Mysticeti [7], lift this idea to smart-contract objects: a transaction on an object owned by a single account uses broadcast, while transactions on objects shared among multiple accounts rely on consensus. In contrast to these works, BLSMR offers a general replication framework to leverage commutativity in distributed applications, and it permits implementing arbitrary (Byzantine) linearizable objects. We discuss how it may also benefit from specific applicative use cases in §3.2.

In some BFT SMR protocols, replicas return optimistically the result of command execution to the client. For instance, this is the case in the Q/U [1] and PGB [79] protocols. Basil [90] is also a recent work that relies on such a mechanism to execute transactions in BFT systems. Instead of SMR, Basil uses deferred update. In the best case, execution takes four message delays: one round-trip to (optimistically) execute the transaction, then another one for commitment. HotStuff-1 is a recent variant [55] that improves on HotStuff's responsiveness by adding a speculative phase and eliminating redundant protocol phases. The authors also describe a slotting mechanism to rotate leaders. HotStuff-2 [68] reduces the number of phases of HotStuff to two. Like HotStuff-1, it rotates its leader for performance and needs at least three message delays to commit a command. HotStuff-1 and Proof-of-Execution (PoE) [51] lower latency with speculative execution: replicas execute a transaction and reply before it is committed, so this execution may later have to be rolled back. In contrast, a command that takes the fast path of Wintermute is committed, and its execution is final. We evaluate HotStuff against our approach in §6. Optimistic execution can be added to the protocols in §5. However, it is not always applicable because it necessitates that side-effects can be rolled back.

We can deconstruct existing leaderless solutions (e.g., the ones in Table 1) with the help of the abstract machine de-

[1] Table 1 reports on the performance of DAG-Rider when combined with [17]. In [56], the authors consider amortized message complexity and the impossibility of man-in-the-middle attacks. Under such assumptions, $\mathcal{M}$ and $\mathcal{A}$ reduce respectively to $O(n)$ and $O(1)$.

picted in §4. In the vein of prior works on Paxos [16], this is a valuable exercise to better understand them. However, it is outside of the scope of this paper which focuses on proposing new efficient algorithms. The machine also permits comparing solutions for the fail-stop and Byzantine failure models. Indeed, the trusted assignment operator abstracts away that Algorithm 1 tolerates Byzantine failures. If such assignments are regular ones, the machine supports crash failures.

Trust management. When processes are Byzantine, managing trust is necessary. PBFT [25] and HotStuff [100] manipulate certificates, or quorum certificates. The seminal work in [22] formalizes how these collections of messages are used in a protocol. In particular, the authors explain how to leverage threshold cryptography. This technique is also usable in our protocols to lower the cost of verifying signatures. In [22], the authors introduce the notion of external validity for Byzantine agreement. The work in [27] studies how validity impacts agreement. Validity bounds the effects of Byzantine processes on a computation and allows connecting distributed services. For instance, Tendermint [3] asks each transaction to be validated by an upper applicative layer. Our connection between CONS and DDS works similarly to [88]. In line with these works, the notions introduced in §4.1 offer a convenient framework to describe trust in a distributed computation.

Decomposition. Several works [67, 80, 99] decompose leaderless SMR into a dependency collection service and a consensus service. Our framework in §3 and §4 generalizes this approach to Byzantine systems. As pointed out by Lamport [64], hardening replication protocols to tolerate Byzantine faults is challenging. Some prior leaderless protocols (e.g., ezBFT [5]) are known to be buggy. This shows that achieving such a generalization is not straightforward.

In [35], the authors propose Narwhal, a mempool abstraction which bears similarities to the DDS service. One of the key differences is that DDS takes into account the semantics of the replicated service, by leveraging the commutativity of commands. The work in [86] proposes a leaderless protocol aimed at implementing a digital currency where consensus is only used for transaction conflict resolution whereas transaction dissemination is done via a broadcast protocol. The resulting protocol has a strong resemblance to the decomposition we have proposed (that is, decoupling ordering from agreement via DDS and CONS).

In his seminal work on Generalized Consensus [62], Lamport proposes a framework to abstract several variations of the Paxos protocol. Each protocol is instantiated with the appropriate definition of the c-struct abstraction. In [6], the authors depict a general framework for BFT SMR called Abstract. Abstract allows composing multiple protocols. At a given time, a single protocol runs and the system can move to a different one depending on the load and/or asynchrony conditions.

In [4], the authors propose a definition of leaderlessness for consensus, along with a corresponding protocol. Their definition is based on the synchronous-k round-based model, in which executions are (eventually) synchronous and at most $k < n$ processes can be suspended per round. A protocol is considered leaderless if it can make progress in every synchronous-1 execution. Under this definition, many existing protocols (such as [33]) are not leaderless, since an adversary can prevent their progress. Instead, we follow the more conventional approach, defining leaderlessness in state-machine replication as the lack of a stable leader.

8 Closing Remarks

In what follows, we discuss the limitations of this work and some future research directions, before closing.

Discussion. We now discuss the assumptions and limitations of this work, as well as the trade-offs made by our protocols.

Theory vs. simulation. Our correctness results are established by mathematical proofs for Byzantine faults, whereas our TLA$^+$ specifications were model-checked on small instances and for crash faults only. The fast-path guarantee of Wintermute (Theorem 3) holds in synchronous failure-free runs, and it counts message delays. Our simulations adopt the same idealized view of performance: they model the network latency between replicas, but cryptographic operations, message processing, and (un)marshalling are free (§6.1). Moreover, only the failure-free path of each protocol is implemented. In a real deployment, these costs add to the latency of all the protocols. In Wintermute, the commit message of a command carries the replies of a quorum, that is, $O(n)$ signatures that each replica checks. On the fast path, these replies are identical, so their signatures can be aggregated, e.g., with multi-signatures. Tail latency is also expected to be higher in practice: as observed for other leaderless protocols [93], long chains of dependencies among conflicting commands may delay execution.

Bandwidth. Our protocols are optimized for latency: in Wintermute, each command is broadcast to all the replicas, and each replica replies with its dependencies. This is the classical trade-off in SMR between latency and throughput. DAG-based protocols such as Bullshark decouple data dissemination from ordering and batch commands to sustain a high throughput, at the cost of latency.

Resilience. Wintermute requires $n \geq 5f + 1$ replicas, instead of $3f + 1$ for leader-based protocols such as HotStuff. Thus, for the same number of replicas, it tolerates fewer failures. We believe that this trade-off is acceptable in small permissioned deployments, such as a consortium of known organizations, where latency matters more than the number of tolerated failures. 3Jane tolerates even fewer failures, as $f \leq (\sqrt{n} - 2)/3$ (e.g., $f \leq 14$ when $n = 2{,}025$). This bound holds against an adversary that may choose the faulty processes after the quorums are known. Two facts mitigate it in

practice: *(i)* the quorums used by the DDS service are chosen at random (or, e.g., based on trust), and *(ii)* the processes participating in the consensus instance of a command are selected using a verifiable random function. If the faulty processes are chosen before these random choices, we expect 3Jane to tolerate a larger number of failures with high probability, as in the committees of Algorand [46]. A precise analysis is left for future work.

Failures. Similarly to HotStuff, Wintermute is tailored for small permissioned systems. Its fast path requires the replies of all the replicas. When a replica crashes or slows down, commands still commit, but on the slow path, that is, after four message delays instead of two. Likewise, a Byzantine replica can prevent the fast path by reporting mismatching dependencies. To benefit again from the fast path, the system must be reconfigured to exclude such a replica. Our evaluation covers neither reconfiguration nor these failure scenarios.

Future work. Our system model assumes a static, known-in-advance set of processes, commonly referred to in the literature as permissioned. A more realistic and challenging model is the permissionless setting [73], where an unbounded number of processes may exist, with a finite subset active at any given time (churn), and peers are free to join or leave the system dynamically.

Many of the concepts and algorithms from this work can be adapted to the permissionless setting. One approach involves leveraging a proof-of-stake (PoS) mechanism, as pioneered in [19]. In this approach, each process p has a stake $0 \leq \mathfrak{s}(p) \leq 1$ in a globally available asset, such as a token. This information is maintained within the replicated state machine. Instead of using an upper bound f on the number of Byzantine processes, a quorum system relies on stakes. That is, the intersection properties are built upon the stakes of the members of a quorum, instead of its cardinality—as in the weighted voting algorithm of Gifford [45].

Some state-machine commands permit moving assets among stakeholders. In BLSMR, these commands are non-commuting with other commands. To limit their impact on protocol performance, such asset transfer operations can be deferred—for example, taking effect only at the start of a new epoch, as proposed in [57, 78, 101]. Alternatively, one can build on the Stellar technique [66], in which permissionless quorum systems are constructed from the local views of each peer [44, 94].

A comprehensive presentation of porting BLSMR to a permissionless model is left for future work.

Conclusion. This work presents an abstract machine to construct leaderless BFT SMR protocols in a principled manner, together with three efficient implementations: an approach à la Egalitarian Paxos, a fast protocol (Wintermute), and a protocol tailored for large-scale deployments (3Jane). All of them leverage commutativity. Wintermute executes conflict-free commands in optimal time, that is, after a single round-trip from any replica in the system.

Acknowledgement

This work was partially supported by the REDONDA project, funded by the EU CHIST-ERA network.

Artifacts

Formal Specifications. As mentioned in §1, the algorithms and abstractions presented in this work are specified in TLA$^+$. The specifications are available in an online companion repository [92]. In detail, the repository contains the following TLA$^+$ specifications:

- `SMR.tla`: State-machine replication (§2.2)
- `BLSMR.tla`: Byzantine leaderless state-machine replication (§3)
- `VectorConsensus.tla`: Vector consensus (§4.2)
- `DDS.tla`: Dependency discovery service (§4.2)
- `Machine.tla`: Abstract BLSMR Machine (§4.2)
- `EPaxos.tla`: Egalitarian Paxos (§5.1)
- `Wintermute.tla`: Wintermute (§5.2) and 3Jane (§5.3)

The correctness (safety and liveness) of these specifications was verified for small-scale models in the crash fault-tolerant case.

Simulations. The source code of all simulations, along with instructions and scripts to generate plots, is available in an online companion repository [82].

References

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 59–74, Brighton, United Kingdom, 2005. Association for Computing Machinery. doi:10.1145/1095810.1095817.

[2] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. Revisiting fast practical Byzantine fault tolerance. *CoRR*, abs/1712.01367, 2017. URL: http://arxiv.org/abs/1712.01367, arXiv:1712.01367.

[3] Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. Correctness of Tendermint-core blockchains. In Jiannong Cao, Faith Ellen, Luís Rodrigues, and Bernardo Ferreira, editors, *22nd International Conference on*

Principles of Distributed Systems, OPODIS 2018, December 17-19, 2018, Hong Kong, China, volume 125 of *LIPIcs*, pages 16:1–16:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.OPODIS.2018.16.

[4] Karolos Antoniadis, Julien Benhaim, Antoine Desjardins, Elias Poroma, Vincent Gramoli, Rachid Guerraoui, Gauthier Voron, and Igor Zablotchi. Leaderless consensus. *Journal of Parallel and Distributed Computing*, 176:95–113, 2023. URL: https://www.sciencedirect.com/science/article/pii/S0743731523000151, doi:10.1016/j.jpdc.2023.01.009.

[5] Balaji Arun, Sebastiano Peluso, and Binoy Ravindran. ezBFT: Decentralizing Byzantine fault-tolerant state machine replication. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*, pages 565–577. IEEE, 2019. doi:10.1109/ICDCS.2019.00063.

[6] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. *ACM Trans. Comput. Syst.*, 32(4), January 2015. doi:10.1145/2658994.

[7] Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. Mysticeti: Reaching the latency limits with uncertified DAGs. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025. URL: https://www.ndss-symposium.org/ndss-paper/mysticeti-reaching-the-latency-limits-with-uncertified-dags/.

[8] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. SoK: Consensus in the age of blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, AFT '19, pages 183–198, Zurich, Switzerland, 2019. Association for Computing Machinery. doi:10.1145/3318041.3355458.

[9] Mathieu Baudet, George Danezis, and Alberto Sonnino. FastPay: High-performance Byzantine fault tolerant settlement. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, AFT '20, pages 163–177. ACM, 2020. doi:10.1145/3419614.3423249.

[10] Rida Bazzi and Maurice Herlihy. Clairvoyant state machine replication. *Information and Computation*, 285:104701, 2022. URL: https://www.sciencedirect.com/science/article/pii/S089054012100016X, doi:10.1016/j.ic.2021.104701.

[11] Rida A. Bazzi and Yin Ding. Non-skipping timestamps for Byzantine data storage systems. In Rachid Guerraoui, editor, *Distributed Computing, 18th International Conference, DISC 2004*, volume 3274 of *Lecture Notes in Computer Science*, pages 405–419, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[12] BeaconScan. Ethereum 2.0 Beacon Chain explorer - validator leaderboard, 2026. Accessed: July 2026. URL: https://beaconscan.com/validators.

[13] Nir Bitansky. Verifiable random functions from non-interactive witness-indistinguishable proofs. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography*, pages 567–594, Cham, 2017. Springer International Publishing.

[14] Bitnodes. Global Bitcoin nodes distribution, 2026. Accessed: November 2026. URL: https://bitnodes.io.

[15] Sam Blackshear, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, Brandon Williams, and Lu Zhang. Sui Lutris: A blockchain combining broadcast and consensus. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, pages 2606–2620. ACM, 2024. doi:10.1145/3658644.3670286.

[16] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. Deconstructing Paxos. *SIGACT News*, 34(1):47–67, 2003. doi:10.1145/637437.637447.

[17] Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987. URL: https://www.sciencedirect.com/science/article/pii/089054018790054X, doi:10.1016/0890-5401(87)90054-X.

[18] Manuel Bravo, Gregory V. Chockler, and Alexey Gotsman. Making Byzantine consensus live. *Distributed Comput.*, 35(6):503–532, 2022. doi:10.1007/s00446-022-00432-y.

[19] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018. URL: http://arxiv.org/abs/1807.04938, arXiv:1807.04938.

[20] Vitalik Buterin. Ethereum white paper: A next generation smart contract & decentralized application platform, 2013. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[21] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[22] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pages 524–541, Berlin, Heidelberg, 2001. Springer-Verlag.

[23] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In Pierre Fraigniaud, editor, *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26–29, 2005, Proceedings*, volume 3724 of *Lecture Notes in Computer Science*, pages 503–504. Springer, 2005. doi:10.1007/11561927_42.

[24] Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne-Anne Pignolet, Victor Shoup, and Dominic Williams. Internet Computer consensus. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 81–91. ACM, 2022. doi:10.1145/3519270.3538430.

[25] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, New Orleans, Louisiana, USA, 1999. USENIX Association.

[26] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996. doi:10.1145/226643.226647.

[27] Pierre Civit, Seth Gilbert, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. On the validity of consensus. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*, PODC '23, pages 332–343, Orlando, FL, USA, 2023. Association for Computing Machinery. doi:10.1145/3583668.3594567.

[28] Shir Cohen and Idit Keidar. Tame the Wild with Byzantine Linearizability: Reliable Broadcast, Snapshots, and Asset Transfer. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.DISC.2021.18, doi:10.4230/LIPIcs.DISC.2021.18.

[29] Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020*, pages 26–38. IEEE, 2020.

[30] CometBFT. CometBFT consensus specification, 2026. Accessed: November 2026. URL: https://github.com/cometbft/cometbft/blob/main/spec/consensus/consensus.md.

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

[32] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, page 86, 2016. URL: http://eprint.iacr.org/2016/086.

[33] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient leaderless Byzantine consensus and its application to blockchains. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–8, 2018. doi:10.1109/NCA.2018.8548057.

[34] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Evaluating the Red Belly blockchain. *CoRR*, abs/1812.11747, 2018. URL: http://arxiv.org/abs/1812.11747, arXiv:1812.11747.

[35] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and Tusk: A DAG-based mempool and efficient BFT consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, pages 34–50, Rennes, France, 2022. Association for Computing Machinery. doi:10.1145/3492321.3519594.

[36] DFINITY Foundation. Node providers - ICP dashboard, 2026. Accessed: July 2026. URL: https://dashboard.internetcomputer.org/network/providers.

[37] Vincenzo P. Di Perna, Marco Bernardo, Francesco Fabris, Sebastiao Amaro, Miguel Matos, and Valerio Schiavoni. Impact of network topologies on blockchain performance. In *Proceedings of the 19th ACM International Conference on Distributed and Event-Based Systems*, DEBS '25, pages 122–133, Gothenburg, Sweden, 2025. Association for Computing Machinery. doi:10.1145/3701717.3730540.

[38] Volker Diekert and Grzegorz Rozenberg, editors. *The Book of Traces*. World Scientific, 1995. doi:10.1142/2563.

[39] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for Byzantine agreement. *J. ACM*, 32(1):191–204, January 1985. doi:10.1145/2455.214112.

[40] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988. doi:10.1145/42282.42283.

[41] Michael Eischer and Tobias Distler. Egalitarian Byzantine fault tolerance. In *26th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2021, Perth, Australia, December 1-4, 2021*, pages 1–10. IEEE, 2021. doi:10.1109/PRDC53464.2021.00019.

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

[43] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems (extended version). *CoRR*, abs/2003.11789, 2020. URL: https://arxiv.org/abs/2003.11789, arXiv:2003.11789.

[44] Álvaro García-Pérez and Alexey Gotsman. Federated Byzantine quorum systems. In Jiannong Cao, Faith Ellen, Luís Rodrigues, and Bernardo Ferreira, editors, *22nd International Conference on Principles of Distributed Systems, OPODIS 2018*, volume 125 of *LIPIcs*, pages 17:1–17:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.OPODIS.2018.17.

[45] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, SOSP '79, pages 150–162, Pacific Grove, California, USA, 1979. Association for Computing Machinery. doi:10.1145/800215.806583.

[46] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68. ACM, 2017. doi:10.1145/3132747.3132757.

[47] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. Autobahn: Seamless high speed BFT. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*. ACM, 2024. doi:10.1145/3694715.3695942.

[48] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580, 2019. doi:10.1109/DSN.2019.00063.

[49] Vincent Gramoli, Rachid Guerraoui, Andrei Lebedev, Chris Natoli, and Gauthier Voron. Diablo: A benchmark suite for blockchains. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, pages 540–556, Rome, Italy, 2023. Association for Computing Machinery. doi:10.1145/3552326.3567482.

[50] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 307–316. ACM, 2019. doi:10.1145/3293611.3331589.

[51] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. Proof-of-execution: Reaching consensus through fault-tolerant speculation. In *Proceedings of the 24th International Conference on Extending Database Technology (EDBT)*, pages 301–312, 2021. doi:10.5441/002/edbt.2021.27.

[52] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

[53] Hassan Iqbal, Anand Singh, and Muhammad Shahzad. CALM: Sample dataset. https://github.com/iqbal-h/CALM/tree/main/sample_dataset, 2022.

[54] Hassan Iqbal, Anand Singh, and Muhammad Shahzad. Characterizing the availability and latency in AWS network from the perspective of tenants. *IEEE/ACM Transactions on Networking*, 30(4):1554–1568, 2022. doi:10.1109/TNET.2022.3148701.

[55] Dakai Kang, Suyash Gupta, Dahlia Malkhi, and Mohammad Sadoghi. HotStuff-1: Linear consensus with one-phase speculation. *Proc. ACM Manag. Data*, 3(3), June 2025. doi:10.1145/3725308.

[56] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 165–175, Virtual Event, Italy, 2021. Association for Computing Machinery. doi:10.1145/3465084.3467905.

[57] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing Bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium, USENIX Security 2016*, pages 279–296. USENIX Association, 2016.

[58] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4), January 2010. doi:10.1145/1658357.1658358.

[59] Klaus Kursawe. Optimistic Byzantine agreement. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, SRDS '02, page 352, USA, 2002. IEEE Computer Society.

[60] Petr Kuznetsov, Andrei Tonkikh, and Yan X. Zhang. Revisiting optimal resilience of fast Byzantine consensus. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, pages 343–353, Virtual Event, Italy, 2021. Association for Computing Machinery. doi:10.1145/3465084.3467924.

[61] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. doi:10.1145/279227.279229.

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

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

[64] Leslie Lamport. Byzantizing Paxos by refinement. In David Peleg, editor, *Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings*, volume 6950 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2011. doi:10.1007/978-3-642-24100-0_22.

[65] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982. doi:10.1145/357172.357176.

[66] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with Stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 80–96, Huntsville, Ontario, Canada, 2019. Association for Computing Machinery. doi:10.1145/3341301.3359636.

[67] Giuliano Losa, Sebastiano Peluso, and Binoy Ravindran. Brief announcement: A family of leaderless generalized-consensus algorithms. In George Giakkoupis, editor, *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 345–347. ACM, 2016. doi:10.1145/2933057.2933072.

[68] Dahlia Malkhi and Kartik Nayak. Extended abstract: HotStuff-2: Optimal two-phase responsive BFT. Cryptology ePrint Archive, Paper 2023/397, 2023. URL: https://eprint.iacr.org/2023/397.

[69] Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Comput.*, 11(4):203–213, 1998. doi:10.1007/s004460050050.

[70] Dahlia Malkhi, Michael K. Reiter, and Avishai Wool. The load and availability of Byzantine quorum systems. *SIAM Journal on Computing*, 29(6):1889–1906, 2000. doi:10.1137/S0097539797325235.

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

[72] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 402–411, 2005. doi:10.1109/DSN.2005.48.

[73] Michael Merritt and Gadi Taubenfeld. Computing with infinitely many processes. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, pages 164–178, Berlin, Heidelberg, 2000. Springer-Verlag.

[74] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99*, pages 120–130. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814584.

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

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

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

[78] QuantumMechanic. Proof of stake instead of proof of work, 2011. URL: https://bitcointalk.org/index.php?topic=27787.0.

[79] Pavel Raykov, Nicolas Schiper, and Fernando Pedone. Byzantine fault-tolerance with commutative commands. In Antonio Fernández Anta, Giuseppe Lipari, and Matthieu Roy, editors, *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, volume 7109 of *Lecture Notes in Computer Science*, pages 329–342. Springer, 2011. doi:10.1007/978-3-642-25873-2_23.

[80] Tuanir França Rezende and Pierre Sutra. Leaderless state-machine replication: Specification, properties, limits. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 24:1–24:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.24.

[81] Valerio Schiavoni. AWS all-regions ping dataset. https://doi.org/10.5281/zenodo.15590100, 2025.

[82] Konstantin Shprenger. blsmr: simulations of the BLSMR protocols, July 2026. URL: https://github.com/kshprenger/blsmr.

[83] Konstantin Shprenger, Michael Anoprenko, and Petr Kuznetsov. DScale: a simulation framework for large-scale distributed systems. In *Proceedings of the Workshop on Advanced Tools, Programming Languages, and Platforms for Implementing and Evaluating Algorithms for Distributed Systems*, ApPLIED'26, Royal Holloway, Egham, United Kingdom, 2026. Association for Computing Machinery. doi:10.1145/3820355.3820386.

[84] Nibesh Shrestha and Mohan Kumar. Revisiting EZBFT: A decentralized Byzantine fault tolerant protocol with speculation. *CoRR*, abs/1909.03990, 2019. URL: http://arxiv.org/abs/1909.03990, arXiv:1909.03990.

[85] Nibesh Shrestha and Mohan Kumar. Revisiting hBFT: Speculative Byzantine fault tolerance with minimum cost. *CoRR*, abs/1902.08505, 2019. URL: http://arxiv.org/abs/1902.08505, arXiv:1902.08505.

[86] Jakub Sliwinski, Yann Vonlanthen, and Roger Wattenhofer. Consensus on demand. In Stéphane Devismes, Franck Petit, Karine Altisen, Giuseppe Antonio Di Luna, and Antonio Fernandez Anta, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 13751 of *Lecture Notes in Computer Science*, pages 299–313, Cham, 2022. Springer. doi:10.1007/978-3-031-21017-4_20.

[87] Solana Foundation. Solana analytics — live network dashboard, 2026. Accessed: July 2026. URL: https://solanacompass.com/analytics.

[88] João Sousa and Alysson Bessani. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *Proceedings of the 2012 Ninth European Dependable Computing Conference*, EDCC '12, pages 37–48, USA, 2012. IEEE Computer Society. doi:10.1109/EDCC.2012.32.

[89] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: DAG BFT protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, pages 2705–2718, Los Angeles, CA, USA, 2022. Association for Computing Machinery. doi:10.1145/3548606.3559361.

[90] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up BFT with ACID (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 1–17, Virtual Event, Germany, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483552.

[91] Pierre Sutra. On the correctness of Egalitarian Paxos. *Inf. Process. Lett.*, 156:105901, 2020. URL: https://doi.org/10.1016/j.ipl.2019.105901, doi:10.1016/J.IPL.2019.105901.

[92] Pierre Sutra. bft-leaderless-tla: TLA$^+$ specifications, January 2026. URL: https://github.com/otrack/bft-leaderless-tla.

[93] Sarah Tollman, Seo Jin Park, and John K. Ousterhout. EPaxos revisited. In James Mickens and Renata Teixeira, editors, *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 613–632. USENIX Association, 2021. URL: https://www.usenix.org/conference/nsdi21/presentation/tollman.

[94] Robin Vassantlal, Hasan Heydari, and Alysson Bessani. On the minimal knowledge required for solving Stellar consensus. In *43rd IEEE International Conference on Distributed Computing Systems, ICDCS 2023*. IEEE, 2023.

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

[96] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient Byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2013. doi:10.1109/TC.2011.221.

[97] Marko Vukolic. *Quorum Systems: With Applications to Storage and Consensus*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012. doi:10.2200/S00402ED1V01Y201202DCT009.

[98] Michael Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M. Hellerstein, and Ion Stoica. Bipartisan Paxos: A Modular State Machine Replication Protocol. *arXiv CoRR*, abs/2003.00331, 2020. URL: https://arxiv.org/abs/2003.00331.

[99] Michael J. Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M. Hellerstein, and Ion Stoica. SoK: A generalized multi-leader state machine replication tutorial. *J. Syst. Res.*, 1(1), 2021. doi:10.5070/sr31154817.

[100] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 347–356, Toronto ON, Canada, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331591.

[101] Vlad Zamfir. Introducing Casper "the friendly ghost", 2015. URL: https://blog.ethereum.org/2015/08/01/introducing-casper-friendly-ghost.

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

A Linearizable objects with BLSMR

This section explains how to construct linearizable shared objects with BLSMR. In particular, we prove that this abstraction implements a state-machine replication protocol. Our reasoning and algorithms are based upon the notion of trace [38]. We first introduce some preliminary concepts then detail our results.

A.1 Preliminaries

State machine. A sequential (deterministic) object is specified by the following components: *(i)* a set of states $\mathcal{S}$; *(ii)* an initial state $s_0 \in \mathcal{S}$; *(iii)* a set of commands $\mathcal{C}$ that can be performed on the object; *(iv)* a set of their response values $\mathcal{V}$; and *(v)* a transition function $\tau : \mathcal{S} \times \mathcal{C} \to \mathcal{S} \times \mathcal{V}$. In the following, we use special symbols $\bot$ and $\top$ that do not belong to $\mathcal{V}$. 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 loss of generality, we consider that commands are applicable to every state.

Command words. A *command word* x is a sequence of commands. The empty word is denoted 1 and $\mathcal{C}^*$ 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. In that case, c^i denotes the i-th occurrence and $pos(c^i, x)$ is its position in the word x, with $pos(c^i, x) = 0$ if $c \notin x$. 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 $(c,i) \in \mathsf{cmd}(y)$. The concatenation of two words x and y is denoted xy. We write $\sqsubseteq$ the prefix relation induced by the concatenation operator over $\mathcal{C}^*$. That is, $x \sqsubseteq y$ if and only there exists some word z such that $y = xz$. 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 1. *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 $nil \in \mathcal{V}$, and if x is non-empty then we have:

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

Two commands c and d *commute* if in every state s we have:

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

We write $c \bowtie d$ the fact that c and d do not commute, and $c \not\bowtie d$ otherwise. Two words $x, y \in \mathcal{C}^*$ are *equivalent*, written $x \sim y$, when there exist words $z_1, \ldots, z_{k\geq 1}$ such that $z_1 = x$, $z_k = y$ and for all i, $1 \leq i < k$, there exist words z', z'' and commands $c \not\bowtie d$ 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 2 ([38]). *$x \sim y$ holds iff $\mathsf{cmd}(x) = \mathsf{cmd}(y)$ and for any $c \bowtie d$, $c^i <_x d^j \Leftrightarrow c^i <_y d^j$.*

Lemma 3. *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 $a \not\bowtie b$ 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 $\not\bowtie$. 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. A trace captures a class of words that are equivalent, that is they sort non-commuting commands in the same order. It can be seen as a special case of the notion of c-struct used to define the generalized consensus problem [62]. In detail, 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 $\mathcal{C}^*$ by relation $\sim$. An element in Traces is named a *command trace*. For any $x, y, z \in \mathcal{C}^*$, 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 $\mathcal{C}^*$. It follows that Traces together with the append operator defined as $[x][y] = [xy]$ forms a monoid[2]. 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 [38].

Lemma 4. *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 2, $c^i \in z$. Conversely, if $c^i \in z$ then $c^i \notin x$ and by Lemma 2, $c^i \in y$. Then, by applying again Lemma 2, we deduce that $c^i <_z d^j \Leftrightarrow c^i <_{y\setminus x} d^j$. □

Lemma 5. *If $\mathsf{cmd}(x) \subseteq \mathsf{cmd}(y)$ and for any $c \bowtie d$, $c^i <_y d^j \wedge d^j \in x \Rightarrow c^i <_x d^j$, then $[x] \sqsubseteq [y]$.*

Proof. By Lemma 1, $\mathsf{cmd}(x(y \setminus x)) = \mathsf{cmd}(y)$. Then, choose $c, d \in \mathcal{C}$ with $c \bowtie d$ 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 1. □

Lemma 6. *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 4, $x(y \setminus x) \sim y$. Choose $c^i \in x$. By Lemma 3, $\tau^*(s_0, x(y \setminus x)|_{\leq c^i}).\mathsf{val} = \tau^*(s_0, y|_{\leq c^i}).\mathsf{val}$. Since $c^i \in x$, $c^i \notin (y \setminus x)$ and $x(y \setminus x)|_{\leq c^i} = x|_{\leq c^i}$. □

Histories. A *history* is a sequence of *events* of the form $\mathsf{inv}_i(c)$ or $\mathsf{res}_i(c, v)$, where $i \in \mathbb{P}$, $c \in \mathcal{C}$ and $v \in \mathcal{V}$. 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 $\mathsf{c} \prec_h \mathsf{d}$ the causality relation in history h. That is, $\mathsf{c} \prec_h \mathsf{d}$ holds when the response of c precedes the invocation of command d in 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 $i \in \mathbb{P}$; *(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 $\mathit{complete}(h)$ 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* [52] when it can be extended (by appending zero or more responses) into some history h' such that $\mathit{complete}(h')$ is equivalent to a legal and sequential history l preserving the real-time order in h, i.e., $\prec_{h'} \subseteq \prec_l$.

A.2 Algorithm

Algorithm 4 presents the pseudo-code of our construction atop BLSMR. This construction implements a state-machine replication protocol: in line 13, the process submits command c, and in line 18 the command is executed. At line 18, the

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

Algorithm 4 Linearizable objects with BLSMR– code at process p.

```
1:  Variables:
2:    B                                          // an instance of BLSMR, with c ⋈ d ⇒ c ≻ d
3:    S ← s_0                                    // local copy of the sequential object
4:    L ← 1                                      // the log of executed commands
5:  invoke(c) :=                                 // client
6:    eff: choose q ∈ ℙ; coord(c) ← q
7:         send(c) to q
8:  respond(c) :=
9:    pre: ∃v ∈ 𝒱, ∃Q ⊆ ℙ, (|Q| = f + 1) ∧ (∀q ∈ Q, recv(c, v) from q)
10:   eff: return v
11: invoke(c) :=                                 // replica
12:   pre: recv(c) from client(c)
13:   eff: B.submit(c)                           // SMR.submit(c)
14: execute(c) :=
15:   pre: B.phase(c) = stable ∧ c ∉ L
16:   eff: let β be the largest subset of B.deps*(c) s.t. ∀d ∈ β, B.phase(d) = stable ∧ d ∉ L
17:        forall d ∈ β ordered by →
18:          L ← Ld; (S, v) ← τ(S, d); E ← E ∪ {d}   // SMR.execute(c)
19:          send(d, v) to client(d)
```

process also takes care of returning the result of the command to the caller.

In more detail, Algorithm 4 works as follows. There are two sets of processes: *clients* and *replicas*. A replica is in charge of applying commands against their local copy of the object and answer back to the clients. In Algorithm 4, this corresponds to the logic in lines 11 to 19. A client executes commands on the shared object. This happens in lines 5 to 10. As in [100], clients are correct, and there are finitely many of them.

At a process, Algorithm 4 employs the following variables:

- B is an instance of the BLSMR abstraction. The conflict relation ($\asymp$) is set to an over-approximation of the non-commutativity relation ($\bowtie$) among commands; that is for any two commands c and d, if $\mathsf{c} \bowtie \mathsf{d}$ then $\mathsf{c} \asymp \mathsf{d}$.
- S is a local copy of the state of the object, initially set to s_0.
- L stores the log (a command word) of all the commands executed against the local copy.

Each line of Algorithm 4 is atomic. To apply a command c on the shared object, a client process executes *invoke*(c). This call designates some replica process as the coordinator to which the command is sent (line 7). Upon receiving command c, the coordinator submits it to BLSMR (line 13).

Command c gets executed once it is stable (line 15). Execution takes place at lines 16 to 19. To execute c, a replica process first creates a set of commands, or *batch*, β that execute together with c. Inside a batch, commands are ordered according to the partial order $\rightarrow$ (line 17). Let $<$ be a canonical total order over $\mathcal{C}$, e.g., the lexicographic order. Then, $\mathsf{c} \rightarrow \mathsf{d}$ holds iff *(i)* $\mathsf{c} \in deps^*(\mathsf{d})$ and $\mathsf{d} \notin deps^*(\mathsf{c})$; or *(ii)* $\mathsf{c} \in deps^*(\mathsf{d})$, $\mathsf{d} \in deps^*(\mathsf{c})$ and $\mathsf{c} < \mathsf{d}$. Relation $\rightarrow$ defines the *execution order* at a replica process. If there is a one-way transitive dependency between two commands, Algorithm 4 plays them in that order. Otherwise, the algorithm breaks the tie using the arbitrary order $<$.

Once executed, the result of c is sent back to the client (line 19). The client returns such a response value once it has been received from $f+1$ replicas (line 9).

A.3 Correctness

In what follows, r is a finite run of Algorithm 4. For some variable *var*, we write var_p the value of *var* at replica process p during r. The notation var_p^r refers to the value of var_p at the end of the run r.

The execution mechanism at lines 16 to 19 applies in order the commands in the batch β to update S_p. Such an approach maintains the following invariant:

Proposition 7. *Consider two conflicting commands* c *and* d. *If* $p \in$ *Correct executes* c *before* d *in the same batch, then* $\mathsf{c} \rightarrow \mathsf{d}$ *is true at that moment in time.*

Proof. If c and d execute in the same batch, either *execute*(d) or *execute*(c) takes place at p. In the first case, according to the way a batch is built at line 16, $\mathsf{c} \in deps^*(\mathsf{d})$ holds. Because c executes first, line 17 requires that $c \rightarrow d$ is true. In the second case, applying the same argument, $\mathsf{d} \in deps^*(\mathsf{c})$ and again because c is executed first, $\mathsf{c} \rightarrow \mathsf{d}$ must be true. □

Next, we establish that conflicting commands are executed at the correct processes in a sound order. Given two conflicting commands c and d, $\mathsf{c} \mapsto_p \mathsf{d}$ holds when process p

is correct and it executes c before d in line 18. The global execution order is defined as: $\mapsto = \cup \bigcup_{p \in Correct} \mapsto_p$.

Proposition 8. *$\mapsto$ is irreflexive and asymmetric over $\mathcal{C}$.*

Proof. According to the pseudo-code in lines 14 to 19, a command executes only once at a process. Hence, relation $\mapsto$ is irreflexive.

Next, we establish that $\mapsto$ is asymmetric. That is, for any two conflicting commands c and d, $\mathsf{c} \mapsto_p \mathsf{d} \Rightarrow \mathsf{d} \not\mapsto_q \mathsf{c}$, where p and q are two correct processes. The proof goes by contradiction. Commands c and d are executed respectively by p and q. Let D and D' be respectively the values of $deps(\mathsf{c})$ and $deps(\mathsf{d})$ when the processes execute the commands. Due to the precondition at line 15, the two commands must be committed at that time. By the Consistency property of BLSMR, either $\mathsf{d} \in D$ or $\mathsf{c} \in D'$. Without loss of generality, assume that $\mathsf{d} \in D$ holds. We know that $\mathsf{d} \in deps^*_p(\mathsf{c})$ at the time p executes c in a batch β. Because $\mathsf{c} \mapsto_p \mathsf{d}$, d must also be in β. Applying Proposition 7, $\mathsf{c} \to \mathsf{d}$. As a consequence, $\mathsf{c} \in deps^*_p(\mathsf{d})$ and $\mathsf{c} < \mathsf{d}$. Now consider the point in time where q executes command d. Name β' the corresponding batch. By the Agreement property of BLSMR, $\mathsf{c} \in deps^*_q(\mathsf{d})$ at that time. Consequently, as $\mathsf{d} \mapsto_q \mathsf{c}$, c is in β'. It follows that $d < c$; a contradiction. □

We are now ready to prove our main result.

▷ Theorem 1. BLSMR implements a Byzantine fault-tolerant state-machine replication protocol.

Proof. (**Validity**) Assume that c gets executed at line 18 in Algorithm 4. By the Validity property of BLSMR, c is a command. (**Integrity**) When a command c is executed at line 18, it is added to the log L. Thus, the test in line 16 ($\mathsf{c} \notin L$) ensures that c is executed at most once. (**Consistency**) This follows immediately from Proposition 8. (**Liveness**) Consider that some correct process submits c (line 13), or executes it (line 18), in the run. We prove that $deps^*(\mathsf{c})$ is eventually stable at every correct process q. If this holds then, according to the pseudo-code in lines 15 and 16, q eventually executes command c. The proof is by contradiction. By the proof of Theorem 2, c is eventually committed at q at some point in time, say t_0. Suppose that $deps^*(\mathsf{c})$ is not stable at q at some time $t_1 > t_0$. By the Validity property of BLSMR, all the elements in $deps^*(\mathsf{c})$ are commands. Hence, there must exist a non-committed command $\mathsf{c}_1 \in deps^*(\mathsf{c})$ at time t_1. We claim that c_1 is eventually committed at q. Assume otherwise. Since $\mathsf{c}_1 \in deps^*(\mathsf{c})$, there exists a command d with $\mathsf{c}_1 \in deps(\mathsf{d})$ at q. By the Completeness property of $\mathcal{FD}$, $coord(\mathsf{c}_1)$ is eventually permanently suspected at q. Hence, q eventually submits c_1 (line 7 in Algorithm 1), if it has not done so already. As shown in the proof of Theorem 2, a command submitted at a correct process is eventually committed at every correct process; a contradiction. Thus, c_1 is committed at q at some time $t_2 > t_1$. By repeating the above reasoning, there are infinitely many non-committed commands $(\mathsf{c}_i)_{i \geq 1}$ in the run. For each such command c_i, some client i invoked it. Because clients access sequentially the replicated state machine, this requires infinitely many clients; a contradiction. □

Hereafter, we explain that the response values obtained by the clients are linearizable. We write h the history corresponding to the invocation (line 5) and response (line 10) events in r at the correct clients. The propositions that follow explain how the response values of h are computed.

Proposition 9. $\forall p \in Correct, \Box(\forall c \in \mathcal{C}.\ |L_p|_c \leq 1)$.

Proof. (by induction) L_p is initially equal to 1. Suppose process p appends c to L_p at line 18. According to line 15, this requires $\mathsf{c} \notin L_p$. Hence, from that point in time $|L_p|_c = 1$. □

Proposition 10. $\forall p \in Correct, \Box(S_p = \tau^*(s_0, L_p).\mathsf{st})$.

Proof. (by induction.) Initially $L_p = 1$, leading to $\tau^*(s_0, L_p).\mathsf{st} = s_0$ at start time. This coincides with the initial value of S_p. In line 18, variable S_p is changed to ${S_p}' = \tau^*(S_p, c).\mathsf{st}$, and L_p to $L'_p = L_p c$. By induction, $S_p = \tau^*(s_0, L_p).\mathsf{st}$. It follows that:

$$\begin{aligned} {S_p}' &= \tau^*(S_p, c).\mathsf{st} \\ &= \tau^*(\tau^*(s_0, L_p).\mathsf{st}, c).\mathsf{st} \\ &= \tau^*(s_0, L'_p).\mathsf{st} \end{aligned}$$

□

Proposition 11. $\forall \mathsf{res}_{client(c)}(c, v) \in h, \exists p \in Correct,$ $v = \tau^*(s_0, L^r_p|_{\leq c}).\mathsf{val}$.

Proof. At line 9, $client(c)$ receives $f+1$ times the response value v. This implies that it receives such a value from a correct process, say p. From the pseudo-code in line 19, v is the result of the computation in lines 16 to 19. Let L be the value of L_p before this computation. Applying Proposition 10 leads to $S_p = \tau^*(s_0, L).\mathsf{st}$. Thus, we have $v = \tau^*(s_0, Lc).\mathsf{val}$. By Proposition 9, $Lc = L^r_p|_{\leq c}$. □

Next, we construct a linearization of h. Let E be the commands executed during r, that is $E = \bigcup_{p \in Correct} \mathsf{cmd}(L^r_p)$. Recall that $\prec_h$ denotes the real-time order in which commands are executed in h. We write $\mapsto^*$ the union of the relation $\mapsto$ and $\prec_h$.

Proposition 12. *$\mapsto^*$ is acyclic over E.*

Proof. We prove the following invariant: for any two commands $\mathsf{c}, \mathsf{d} \in E$, if $\mathsf{c} \mapsto^* \mathsf{d}$ then when a correct process executes command d for the first time, another correct process has already executed c. From this invariant, the acyclicity of $\mapsto^*$ over E is immediate.

To show the invariant, there are two cases to consider. (Case $\mathsf{c} \prec_h \mathsf{d}$) By definition of $\prec_h$, at the time d is invoked by a correct client, another correct client has already received a

response value, say v, for c. According to line 9, the response v is received $f+1$ times at that client. Hence, some correct process has already executed c. Because Byzantine processes cannot forge state-machine commands, if a process (faulty or not) executes command d, this must happen after d is invoked. (Case $\mathsf{c} \mapsto \mathsf{d}$) Let p be the correct process that executes first command d. Because $\mathsf{c} \mapsto \mathsf{d}$ holds, there exists another correct process q such that $\mathsf{c} \mapsto_q \mathsf{d}$. According to Proposition 8, $\mathsf{d} \not\mapsto_p \mathsf{c}$. Since p executes d, $\mathsf{c} \mapsto_p \mathsf{d}$. Hence, at the time it executes d, p has already executed c. □

To linearize history h, we use a sequence δ. This sequence can be any linear extension of $\mapsto^*$ over the set E.

Proposition 13. $\forall p \in \mathit{Correct}, [L_p^r] \sqsubseteq [\delta]$.

Proof. For every correct process p, $\mathsf{cmd}(L_p^r) \subseteq \mathsf{cmd}(\delta)$. Consider a pair of non-commuting commands c and d in δ, with $\mathsf{c} <_\delta \mathsf{d}$ and $\mathsf{d} \in L_p^r$. Observe that if $\mathsf{c} \notin L_p^r$ or $\mathsf{d} <_{L_p^r} \mathsf{c}$, then $\mathsf{d} \mapsto_p \mathsf{c}$. This contradicts the definition of δ. Thus, $\mathsf{c} <_{L_p^r} \mathsf{d}$. Applying Lemma 5, $[L_p^r] \sqsubseteq [\delta]$. □

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. Let h' be the resulting history that, by construction, completes h.

Proposition 14. $l \sim h'$

Proof. By applying Proposition 11, Proposition 13, and Lemma 6. □

Proposition 15. $\prec_{h'} \subseteq \prec_l$

Proof. By definition of the sequence δ from which l is built. □

Theorem 16. *For every finite run r of Algorithm 4, the history h induced by r is linearizable.*

Proof. The result follows from Propositions 14 and 15. □

B Byzantine Quorum Systems

Byzantine quorum systems are standard mechanisms to hide faulty processes in BFT SMR. They extend quorum systems (from the crash-failure model) to deal with Byzantine failures. Below, we present this notion following standard definitions [69, 97]. We also introduce a new type of quorum systems, the witnessing Byzantine quorum system, that are used in §5.

B.1 Definitions

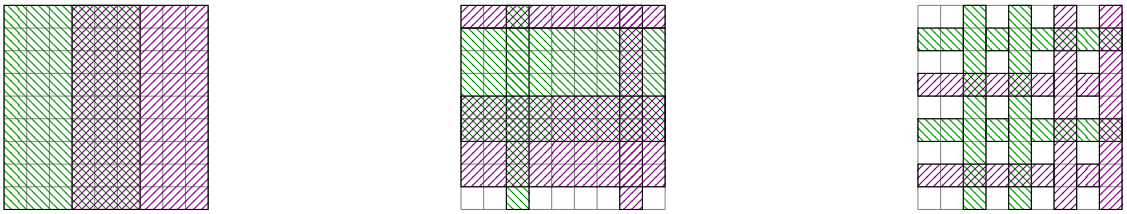
A quorum system $\mathfrak{Q} \subseteq 2^{\mathbb{P}}$ is a non-empty set of subsets of $\mathbb{P}$, every pair of which intersects. Each $Q \in \mathfrak{Q}$ is called a quorum. Byzantine quorum systems are specified given an adversary (also called a fail-prone system). An adversary $\mathcal{B} \subseteq 2^{\mathbb{P}}$ defines all the failures that may occur in some execution of the system (that is, for any run r, $\mathit{faulty}(r) \in \mathcal{B}$). This set is closed by inclusion, i.e., $B \in \mathcal{B}$ and $B' \subseteq B$ then $B' \in \mathcal{B}$. For instance, the $n \geq 3f+1$ assumption captures the common threshold adversary model, where $\mathcal{B} = \{B \subseteq \mathbb{P} : |B| \leq f\}$. A *Byzantine quorum system* (BQS) is a quorum system together with an adversary. As usual, we consider that quorums are closed under supersets: if Q is a quorum and $Q \subseteq Q' \subseteq \mathbb{P}$, then Q' is also a quorum. This is without loss of generality, since enlarging a quorum only enlarges its intersections with the other quorums, and thus preserves all the properties below. In particular, $\mathbb{P}$ is always a quorum. When we describe a quorum system, we only list its minimal quorums.

A base BQS is the *dissemination quorum system*, or DQS for short. Such a quorum system is defined through the following properties: (D-Consistency) $\forall Q_1, Q_2 \in \mathfrak{Q}, \forall B \in \mathcal{B}, Q_1 \cap Q_2 \not\subseteq B$ (Availability) $\forall B \in \mathcal{B}, \exists Q \in \mathfrak{Q}, B \cap Q = \varnothing$ When (at most) f failures occur, D-Consistency ensures that any two quorums must intersect in a correct process. Availability guarantees the existence of a quorum containing only correct processes. In the threshold adversary model with $n \geq 3f+1$, a base example of such a quorum system consists of all the sets of $\lceil (n+f+1)/2 \rceil$ processes [69]. When $n = 3f+1$, these are the sets of $2f+1$ processes. Figure 7a depicts two such quorums for $n = 9^2$ and $f = 26$: each quorum contains 54 processes (six columns), and the two quorums intersect over $f+1 = 27$ processes.

A *masking quorum system* (MQS) is a restricted form of dissemination quorum system. D-consistency is replaced with the following stronger property: (M-Consistency) $\forall Q_1, Q_2 \in \mathfrak{Q}, \forall B_1, B_2 \in \mathcal{B} : (Q_1 \cap Q_2) \setminus B_1 \not\subseteq B_2$ This guarantees that any two quorums intersect over $2f+1$ processes, and therefore over (at least) $f+1$ correct processes. In the threshold adversary model with $n \geq 4f+1$, all the sets of $\lceil (n+2f+1)/2 \rceil$ processes form an MQS [69]. Another type of MQS is a grid quorum system. It consists of a column together with $2f+1$ rows. Figure 7b depicts such a system when $f = 2$.

B.2 Witnessing quorum systems

Dissemination and masking quorum systems are used in several works to construct (safe and atomic) registers [97]. Figure 7c illustrates a novel type of Byzantine quorum systems called *witnessing quorum system* (WQS). A WQS is a special case of masking quorum system. In detail, recall that $\mathfrak{Q}$ denotes the quorum system and $\mathcal{B}$ the adversary. WQS replaces the M-Consistency property of MQS with the following one: (W-Consistency) $\forall Q_1, Q_2 \in \mathfrak{Q}, \forall B_1, B_2, B_3 \in \mathcal{B} :$



(a) (b) (c)

Figure 7: Illustrating quorums in (a) DQS, (b) MQS and (c) WQS quorum systems.

$(Q_1 \cap Q_2) \setminus (B_1 \cup B_2) \not\subseteq B_3$ In the threshold adversary model, this property states that any two quorums intersect over $3f+1$ processes, and thus among them (at least) $2f+1$ are correct ones.

A base example of witnessing quorum system consists of $n = 5f+1$ processes, among which any subset of (at least) $4f+1$ processes is a quorum. Another example is illustrated in Figure 7c. This quorum system is built as a variation of the *M-grid quorum* system proposed in [70]. Processes are arranged into a $\sqrt{n} \times \sqrt{n}$ grid, assuming $f \leq (\sqrt{n}-2)/3$. 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 of ζ rows and ζ columns, with $\zeta = \lceil \sqrt{3f/2+1} \rceil$. In other words, the quorum system is defined as:

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

In Figure 7c, we have $n = 9^2$ and $f = 2$, leading to $\zeta = 2$. Hence, a quorum consists of two rows plus two columns. The propositions below establish the correctness of such a construction, as well as the load of this new quorum system.

Proposition 17. *$\mathfrak{Q}$ is a witnessing quorum system for $f \leq (\sqrt{n}-2)/3$.*

Proof. Consider $Q_1, Q_2 \in \mathfrak{Q}$. We first establish that W-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 Availability, observe that $\sqrt{n} - f \geq 3f+2-f \geq 2f \geq \zeta$. As a consequence, for any $B \subseteq \mathbb{P}$ such that $|B| = f$, we may pick ζ rows and ζ columns whose union does not contain any element in B. □

Proposition 18. *The load of $\mathfrak{Q}$ is $O(\sqrt{f/n})$ when quorums are picked uniformly at random.*

Proof. All the minimal quorums in $\mathfrak{Q}$ have the same size $c(\mathfrak{Q}) = 2 \times \zeta \times \sqrt{n} - \zeta^2$, and, by symmetry of the grid, each process belongs to the same number of them. As quorums are picked uniformly at random, each process is in the picked quorum with probability $c(\mathfrak{Q})/n$, which matches the lower bound of [70]. Hence, the load of $\mathfrak{Q}$ is $(2 \times \zeta \times \sqrt{n} - \zeta^2)/n \in O(\sqrt{f/n})$. □

C Generalized Threshold Union

Consider that $(\mathcal{L}, \sqsubseteq)$ is a lattice and let $(l_j)_{j \in J}$ be some bag of elements in $\mathcal{L}$. The generalized threshold union operator is defined as follows: $\bigsqcup\!\!\!\!\bigsqcup_{j \in J}^{f} l_j = \sqcup \{e : count(e) \geq f+1\}$, where $count(e) = |\{j \in J : e \sqsubseteq l_j\}|$. In other words, the operator $\bigsqcup\!\!\!\!\bigsqcup_{j \in J}^{f} l_j$ returns the least upper bound of all the elements in the lattice L that are smaller than at least $f+1$ elements in the bag $(l_j)_{j \in J}$.

Examples. To illustrate the above definition, assume $(l_j)_{j \in J}$ are the dependencies reported by J. Then $\bigsqcup\!\!\!\!\bigsqcup_{j \in J}^{f} deps_j$ are all the commands reported at least $f+1$ times. In which case, $(\mathcal{L}, \sqsubseteq)$ equals $(2^{\mathcal{C}}, \subseteq)$. This is the original usage in [43].

Another example is provided in §5.2.1, where we use compaction over dependencies. In that case, $\mathcal{L}$ are all the vectors in $\mathbb{N}^{\mathbb{P}}$. Relation $\sqsubseteq$ is the standard comparator such that a vector e is lower than a vector e' iff it is lower than e' across all its dimensions.

D Verifiable Sequencing

In Wintermute, sequence numbers are assigned by clients and dependencies are vectors indexed by clients (§5.2.1). To further compact dependencies as vectors of size $O(n)$, the computation of sequence numbers may be delegated to the coordinators. Unfortunately, if the coordinator is Byzantine, it may assign the same sequence number twice, or exhaust sequence numbers on purpose. To remedy this problem, sequence numbers are generated using a verifiable sequencing service, SEQ. Given some command c, this service offers a single operation *next*(c) returning a sequence number c.*sn*. A call to the service offers the following properties: (Uniqueness) If SEQ returns s and s', then $s \neq s'$. (Monotonicity) If SEQ returns s and then s', then necessarily $s < s'$. (Gap-free) If SEQ returns s and immediately after s', then there is no s'' such that $s < s'' < s'$.

Non-skipping timestamps [11] implement a verifiable sequencing service. In a nutshell, for each process p, a set of $4f+1$ processes is in charge of maintaining the service. Each of these processes holds a clock, initially equal to 0. When executing *next*(c), p queries the service processes and awaits $3f+1$ replies containing their clock values. Process p then takes the $(f+1)$-th largest value among the replies and adds 1 to it. To make the service verifiable, the replies are stored in the proof attached to the sequence number, and the verifier $check_{\mathsf{SEQ}}$ simply redoes the above computation. Before returning, p sends the sequence number to the service processes, which update their clocks accordingly. This last phase can be piggybacked on the next call to the service. Hence, the service costs one round-trip to $4f+1$ processes. This request is executed in parallel with the announcement of c, so it does not add message delays. Using this approach, dependencies are encoded as a compact representation (vector) for commands having verified sequence numbers, together with an explicit list for the other commands. (If available, trusted hardware [96] may also implement this service.)

E Correctness

E.1 Algorithm 1 (BLSMR Machine)

Function $1_{\mathcal{C}}(x)$ is verifiable and indicates if its input x is a state-machine command. Formally, it is defined as: $1_{\mathcal{C}}(x) \triangleq x$ **if** $x \in \mathcal{C}$ **else error**. Based on this definition, we characterize the necessary trust to execute successfully the assignment in line 16 of Algorithm 1.

Proposition 19. *If* $check_{1_{\mathcal{C}}}(_,\mathsf{c})$ *returns true then* c *is a state-machine command.*

Proof. From the definition of a verifier, if $check_{1_{\mathcal{C}}}(_,\mathsf{c})$ evaluates to *true* then $1_{\mathcal{C}}(_)$ returns c. Since no error is raised, c must be a command. □

Proposition 20. *Consider that a correct process p executes the trusted assignment in line 16 in a run r, that is* $deps(\mathsf{c}) \xleftarrow{v,(\mathsf{c},D,b)} D$ *for some* c. *If this assignment does not raise an error then* (i) c *is a command,* (ii) DDS *is called with input* c *and returns (D,b) in r, and* (iii) *either $b = true$, or the service* $\mathsf{CONS_c}$ *returns D in r.*

Proof. If the trusted assignment does not raise an error, then $check_v((\mathsf{c},D,b),D)$ evaluates to *true* at p. Recall that the verifier $check_v$ is defined as:

$$check_v((\mathsf{c},D,b),D) = \textbf{if } b \textbf{ then } check_{\mathsf{DDS}}(\mathsf{c},(D,b)) \textbf{ else } check_{\mathsf{CONS_c}}(D,D)$$

First, suppose $b = true$. In light of the definition above, the verifier $check_{\mathsf{DDS}}(\mathsf{c},(D,b))$ evaluates to *true*. Thus, DDS is called with input c and returns (D,b) in run r. Service DDS is trusted wrt. $(1_{\mathcal{C}},_)$ and as a consequence, it is verifying wrt. $(1_{\mathcal{C}},_)$. It follows that $check_{1_{\mathcal{C}}}(_,\mathsf{c})$ also evaluates to true. Applying Proposition 19, c is a command.

Conversely, assume that $b = false$. In that case, by the definition of $check_v$, $check_{\mathsf{CONS_c}}(D,D)$ evaluates to *true*. Because $\mathsf{CONS_c}$ is verifiable, $propose(D) = D$ occurs in run r. Since $\mathsf{CONS_c}$ is trusted wrt. $(\mathsf{DDS},\mathsf{c})$, $check_{\mathsf{DDS}}(\mathsf{c},(D,_))$ returns *true*. From this, we conclude as in the previous case that c is a command. □

▷ Theorem 2. Algorithm 1 implements BLSMR.

Proof. Suppose a run r of Algorithm 1. We prove that all the properties of BLSMR hold in r.

(Validity) If c is committed at some correct process p, then $deps(\mathsf{c}) \neq \bot$ holds at p. The dependencies of c are modified when p executes successfully line 16. Applying Proposition 20, c is a command.

(Consistency) Let p and q be two correct processes that commit respectively two conflicting commands c and d, with dependencies $D, D' \in 2^{\mathcal{C}}$. Applying Proposition 20 at p and at q, we know that DDS returns D and D' when called respectively with command c and d in r. As DDS is verifiable, $check_{\mathsf{DDS}}$ evaluates to true for both outputs. By the Visibility property, necessarily either $\mathsf{c} \in D'$, or $\mathsf{d} \in D$, holds.

(Agreement) Consider a command c and two correct processes p and q. Command c is committed in line 16. This operation uses the tuple extracted from the payload of the message received at line 14. Let respectively (D,b) and (D',b') be the value of this tuple at p and q when this happens. Applying Proposition 20, DDS.*announce*(c) returns (D,b) in r. Similarly, from the fact that q commits value D' for $deps(\mathsf{c})$, DDS.*announce*(c) returns (D',b') in r. As DDS is verifiable, $check_{\mathsf{DDS}}$ evaluates to true for both outputs. Several cases need to be considered here: (Case $b = true$, or $b' = true$.) By the Weak Agreement property, $D = D'$. (Case $b = b' = false$.) Applying Proposition 20, $\mathsf{CONS_c}$ returns D in r. Similarly, $\mathsf{CONS_c}$ returns D' in r. By the Agreement property of consensus, $D = D'$.

(Liveness) Suppose some command c is submitted at a correct process p in r. We show that c is eventually committed at every correct process q. For this, we observe that if q receives a tuple (c,D,b) at line 14 and the assignment at line 16 succeeds, q commits c. Hence we have to prove that one such tuple exists.

Both CONS and DDS are responsive services, that is any call to these services at a correct process eventually returns. It follows that the calls in lines 10 and 11 return at process p. Thus, p eventually executes line 12. Since links are reliable, process q eventually receives a tuple (c,D,b) from p.

If b equals *true*, then (D,b) is the value p returned at line 10 from calling DDS.*announce*(c) in r. Since DDS is verifiable, the verifier $check_{\mathsf{DDS}}(\mathsf{c},(D,b))$ evaluates to *true* at q. As a

consequence, $check_v((\mathsf{c},D,b),D)$ evaluates to *true* and the assignment at line 16 succeeds.

Alternatively, assume that b equals *false*. Because process p is correct, D is the response value to the call $\mathsf{CONS_c}.propose(_)$ at line 11. The Validity property of consensus implies that D is proposed by some process. Consider some linearization of r in which this call returns. As $\mathsf{CONS_c}$ is verifiable and $\mathsf{CONS_c}.propose(D) = D$ in (some linearization of) r, $check_{\mathsf{CONS_c}}(D,D)$ evaluates to *true* at q. It follows that the assignment in line 16 also succeeds in this case.

Finally, suppose that c is committed at some correct process p with a tuple (c,D,b). Process p then sends this tuple to all (line 17). Since links are reliable, every correct process q eventually receives it. The assignment at line 16 succeeds at p, that is, $check_v((\mathsf{c},D,b),D)$ evaluates to *true* at p. By the definition of a verifier, this also holds at q, and q commits c (if it has not already done so). □

E.2 Algorithm 2 (à la EPaxos)

Proposition 1. *Algorithm 2 implements a trusted DDS service.*

Proof. Consider some run r of Algorithm 2.

(Validity) Consider a pair $(D,_)$ such that $check(\mathsf{c},(D,_))$ evaluates to true. According to line 19, D is the union of the dependencies $(D_j)_{j\in Q}$ stored in its proof, and $D \subseteq \mathcal{C}$. Hence, every $\mathsf{d} \in D$ is a command.

(Visibility) Let c and d be two conflicting commands announced in r. Consider two pairs $(D,_)$ and $(D',_)$ such that $check(\mathsf{c},(D,_))$ and $check(\mathsf{d},(D',_))$ both evaluate to true. According to lines 16 to 19, D is the union of the dependencies $(D_j)_{j\in Q_\mathsf{c}}$ that the processes of some quorum Q_c replied to the announcement of c. Similarly, D' is the union of the dependencies replied by the processes of some quorum Q_d to the announcement of d. Since signatures cannot be forged, the reply of a correct process stored in such a proof is the one it actually sent at line 14. From the D-Consistency property of the *BQS* service, the quorums Q_c and Q_d intersect over at least one correct process. Hence, at least one correct process, say j, executes lines 10 to 13 for the two commands. Without loss of generality, consider that these lines are executed first for command c at j. At the time j executes line 13 for command d, $\mathsf{c} \in L$. Hence, j replies to the announcement of d with a set of dependencies including c. According to line 19, $\mathsf{c} \in D'$ holds.

(Weak agreement) Trivial, as *check* rejects any output with $b = true$ (line 20).

(Liveness) Let B be the (Byzantine) failure pattern of r and p a correct process announcing a command. By the Availability property of *BQS*, there exists a Q_0 such that $B \cap Q_0 = \{\}$. Process p broadcasts a message to all the processes including Q_0 in line 5. As links are reliable and these processes are not in B, they eventually deliver the message and each returns a set of dependencies to p (lines 10 to 14). Thus, there exists a time after which the condition at line 6 holds for Q_0 (or for some other quorum). It follows that p eventually returns a value at line 9.

(*Trust*) Let h be the history of Algorithm 2 in r. Suppose a process p invokes $check(\mathsf{c},(D,b))$ and returns *true* in h'. At process p, the following facts are verified: Let Q be the set of processes defined line 16. At line 17, p verifies that Q is a Byzantine quorum. According to line 18, for some announcer process q, every process in Q replies to the announcement of c made by q. At line 19, p re-calculates the union of dependencies, to guarantee that q did not omit (or add) anything. It also ensures that D only contains commands. Further, at line 20, p verifies that the fast path is forbidden. If process q is correct then, by Liveness, q returns (D,b) from the announcement of c in h'. Otherwise, in case q is faulty, we add to h' the following two events: the invocation of $announce_q(\mathsf{c})$ and a matching response that returns (D,b). These events are added at the time in r the first and last process in Q replies to q, respectively. From what precedes, Algorithm 2 outputs (D,b) for the announcement of c in h', as required by function *check*. Moreover $h|Correct = h'|Correct$. The above steps are repeated for every such process p. This reasoning establishes that the DDS service implemented by Algorithm 2 is verifiable. In addition, we note that Algorithm 2 is verifying wrt. $(1_\mathcal{C},_)$: in line 12, a correct process checks that c is a command before replying to the announcer. By D-Consistency, at least one such process is in the quorum used by the announcer. □

E.3 Algorithm 3 (Wintermute)

Lemma 21. *If a correct process executes $reply(\mathsf{c})$ multiple times, it always sends the same dependency set $deps(c)$ in line 14.*

Proof. This lemma follows immediately from the property of the local log. In detail, consider that the process executes $reply(\mathsf{c})$ for the first time, sending some dependency set D. D is computed in line 13. If $\mathsf{d} \in D$, by definition of $L.conflicts(c)$, d precedes c in L and d conflicts with c. As a consequence, at any later time, d is also included in the dependency set computed in line 13. Conversely, if some command e is included at some later point, it must precede c in L. Thus, e is also in D. □

Proposition 2. *Algorithm 3 implements a trusted DDS service.*

Proof. Consider some run r of Algorithm 3.

(Validity) Consider a pair $(D,_)$ such that $check(\mathsf{c},(D,_))$ evaluates to true. According to lines 18 and 19, D is the threshold union of the signed replies $(D_j)_{j\in Q}$ of some quorum Q. Pick $\mathsf{d} \in D$. By definition of the threshold union operator, d is reported by $f+1$ processes. Hence, one of these is correct.

This process only returns d if it is present in the log (line 13). In such a case, due to line 12, d must be a command.

(Visibility) Let a and b be two conflicting commands submitted in run r. Consider two pairs $(D,_)$ and $(D',_)$ such that $check(\mathsf{a},(D,_))$ and $check(\mathsf{b},(D',_))$ both evaluate to true.

According to lines 16 to 19, D is the threshold union of the dependencies $(D_j)_{j\in Q_\mathsf{a}}$ that the processes of some quorum Q_a replied to the announcement of a. Similarly, D' is computed from the replies of some quorum Q_b to the announcement of b. Since signatures cannot be forged, the reply of a correct process stored in such a proof is the one it actually sent at line 14.

By the W-Consistency property of WQS, the quorums Q_a and Q_b intersect over at least $2f+1$ correct processes. These processes all execute lines 10 to 14. The handler is atomic. Hence, either $f+1$ such processes execute these lines for a before b, or vice-versa. Without loss of generality, consider that this is the latter case that happens during r. For each of these processes j, $\mathsf{b}\in deps_j(\mathsf{a})$ holds. Hence, b is reported $f+1$ times in $(deps_j)_{j\in Q_\mathsf{a}}$. Consequently, applying the definition of the threshold union operator, we have $\mathsf{b}\in D$, as required.

(Weak agreement) Consider two pairs $(D,true)$ and $(D',_)$ such that $check(\mathsf{c},(D,true))$ and $check(\mathsf{c},(D',_))$ both evaluate to true. Let Q and Q' be the two quorums stored in their respective proofs (line 16). As above, the reply of a correct process stored in such a proof is the one it actually sent.

($D\subseteq D'$) Let d be some command in D. According to lines 20 and 25, $\mathsf{d}\in D_j$ for any process $j\in Q$. In Algorithm 3, variable *BQS* is a witnessing quorum system. Since $Q=\mathbb{P}$ (from line 24), $Q\cap Q'=Q'$. By the W-Consistency property of WQS (applied to Q' with itself), Q' contains at least $2f+1$ correct processes. By Lemma 21, because these processes are all correct, they send the exact same reply to every announcement of c. In particular, all these processes report d in the proof of D'. It follows that d appears $f+1$ times in $(D_j)_{j\in Q'}$, and it is included in $\lfloor f\rfloor_{j\in Q'} D_j$. Consequently, according to line 19, D' contains command d, as required.

($D'\subseteq D$) Consider some command $\mathsf{d}\in D'$. Command d is reported by $f+1$ processes in the proof of D' (line 19). From which it follows that one of them, say j, is correct. The set Q contains j because it includes all the processes (line 24). By Lemma 21, j reports always the same dependency set. The predicate at line 25 requires that $D=D_j$. It follows that $\mathsf{d}\in D$.

(Liveness) The proof is similar to the one above for Proposition 1. It relies on the Availability property of the WQS quorum system (instead of the Availability of the DQS quorum system earlier).

(Trust) This is analogous to Proposition 1. The two differences are that *(i)* at line 19, the verifier calculates the threshold union of the dependencies returned by the processes in the quorum instead of their union, and *(ii)* at line 20 the fast path is verified. □

F Performance of the Protocols

▷ Theorem 3. Wintermute is a fast Byzantine fault-tolerant state-machine replication protocol.

Proof. From Proposition 2, Algorithm 3 is a trusted DDS service. Applying Theorem 2, it follows that Wintermute implements BLSMR. Consequently, from Theorem 1, Wintermute is a state-machine replication protocol. In what follows, we establish that the Wintermute protocol is fast.

Suppose a conflict-free command c submitted at a process p during a run r. Command c is announced at line 10 in Algorithm 1. According to Algorithm 3, this call returns some set D after 2Δ. Let $\mathsf{d}\in D$ be a command returned from this call. According to line 7, d is reported at least $f+1$ times by the processes in Q. Hence, one of these processes, say p', is correct. It follows that command d results from the computation of line 13 at process p', which implies that d conflicts with c. Since c is conflict-free and all the processes are correct, d is executed everywhere at the time c is submitted, or the converse holds. Suppose the latter case is true. Because $\mathsf{d}\in D$, process p' returns $\mathsf{d}\in deps(\mathsf{c})$ at line 13 in Algorithm 3. At the time d is submitted, c is already executed at p'. This implies that c precedes d in the log of p'. However, a log contains (at most) a single occurrence of each command; contradiction.

From what precedes, every command in D is already executed at every process at the time c is submitted. Hence, all the processes in Q return the set of dependencies D. Since the run is failure-free and synchronous, all the processes are correct. They receive c at the beginning of the round that follows its broadcast (line 5), and they reply immediately. Hence, all the replies are delivered to p at the same time, at the beginning of the next round, and no reply is delivered to p before. Process p receives all of them before taking any other step. Thus, when the condition at line 6 holds for the first time, p has received a reply from every process. Because p takes the largest responding quorum (see line 6), quorum Q contains all the processes in the system (recall that $\mathbb{P}$ is a quorum, see Appendix B.1). It follows that the predicate $fast(\mathsf{c},D)$ evaluates to true at p. Consequently, command c takes the fast path at p.

From what precedes, every command $\mathsf{d}\in D$ is already executed at p. As p receives its own message at line 12 immediately, it follows that c is stable and executes right away at p after 2Δ. This shows that the protocol is fast. □

Proposition 3. *The load of 3Jane is* $O(\sqrt{f/n})$.

Proof. Consider some command c ordered by 3Jane. According to Algorithm 1, c is first processed at the coordinator, then it goes to DDS, and to CONS. It follows that the load of 3Jane is given by:

$$\mathcal{D}_{coord}+(1-\mathcal{D}_{coord})\cdot\mathcal{D}_{\mathsf{DDS}} + (1-\mathcal{D}_{coord})\cdot(1-\mathcal{D}_{\mathsf{DDS}})\cdot\mathcal{D}_{\mathsf{CONS_c}}$$

where $\mathcal{D}_{coord}$ is the load of being the coordinator, $\mathcal{D}_{\mathsf{DDS}}$ is the load of the DDS service and $\mathcal{D}_{\mathsf{CONS_c}}$ is the load of the CONS service. The coordinator is chosen uniformly at random. Hence, $\mathcal{D}_{coord} = 1/n$. In 3Jane, c gets assigned a quorum Q_{c} picked uniformly at random (§5.3). To announce c, the coordinator sends c only to Q_{c}, instead of all the processes (line 5 in Algorithm 3). Hence, $\mathcal{D}_{\mathsf{DDS}}$ is exactly the load of the underlying Byzantine quorum system. From Proposition 18, $\mathcal{D}_{\mathsf{DDS}} = O(\sqrt{f/n})$. There are $3f+1$ processes running instance $\mathsf{CONS_c}$ (§5.3). Such processes are also chosen at random (§5.3). Hence, the load of $\mathsf{CONS_c}$ is $O(f/n)$. From what precedes, the load of 3Jane is $O(\sqrt{f/n})$. □