Fast State-Machine Replication Protocols for Byzantine Systems and Blockchain

Tuanir França Rezende[1] and Pierre Sutra[2]

[1]CEA, France
[2]Telecom SudParis & INRIA, France

Abstract

State-machine replication (SMR) hardened for Byzantine fault-tolerance (BFT) has gained a lot of traction over the past decade. This comes from the fact that such protocols can implement blockchain efficiently. SMR protocols commonly rely on a leader replica to order submitted commands. Some solutions 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 SMR distributes the load and avoids the potential problems due to a Byzantine leader. It can also leverage commutativity at the application level for better performance. Unfortunately to date, existing leaderless solutions are either not fast, or they slow down as soon as a single pair of non-commuting commands is detected. In this paper, we tackle such a concern. We present protocols that execute conflict-free commands, that is commands which do not encounter concurrent non-commuting commands, after a single round-trip to a nearby quorum. To the best of our knowledge, this is the first time a BFT SMR protocol has attained this lower bound. In total, we present three leaderless protocols: *(i)* a Byzantine fault-tolerant variation of the Egalitarian Paxos protocol, *(ii)* a graph-based protocol, and *(iii)* a timestamp-based protocol. All these protocols are written within a common framework. The framework is an abstract distributed machine that implements leaderless state-machine replication for Byzantine systems in a principled manner.

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 rely on proof-of-work (PoW) to perform data replication [16, 56]. This approach offers a high level of security but it has also 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) [4, 15, 19, 25, 34, 36, 48, 73]. 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.

Motivation. In the best case, a state-machine command executes after two message delays, that is one round-trip to the replicas [44]. The design of such BFT SMR protocols is the subject of extensive research [1, 6, 37, 40]. Because SMR reduces to consensus, this effort also exists for agreement protocols [2, 41, 42, 53]. Most of the solutions rely on a central leader replica. If a command is not submitted at the leader replica, its execution requires to wait additional message delays.

Another class of SMR protocols does not employ a leader but instead decentralizes the task of ordering commands. These solutions are generally referred to as leaderless. Leaderless SMR

Protocol	Metrics L	M	A	D	R	Leaderless	Commut.
PBFT [21]	3+1	$O(n^2)$	$O(n^2)$	1	$f < n/3$	×	×
Tendermint [15]	3+1	$O(n^2)$	$O(n^2)$	1	$f < n/3$	×	×
HotStuff [14, 73]	6+1	$O(n)$	$O(n)$	1	$f < n/3$	×	×
FAB Paxos [53]	2+1	$O(n^2)$	$O(n^2)$	1	$f < n/3$	×	×
Quorum [6]	5	$O(n)$	$O(n)$	1	$f < n/3$	√	×
DAG-Rider [39]	12	$O(n^3)$	$O(n^3)$	1	$f < n/3$	√	×
Bullshark [65]	4	$O(n^2)$	$O(n^2)$	1	$f < n/3$	√	×
Mysticeti [7]	3	$O(n^2)$	$O(n^2)$	1	$f < n/3$	√	×
PGB [58]	5	$O(n^2)$	$O(n)$	1	$f < n/5$	√	√
Byblos [10]	5	$O(n^2)$	$O(n^2)$	1	$f < n/4$	√	√
ISOS [30]	3	$O(n^2)$	$O(n^2)$	1	$f < n/3$	√	√
á la EPaxos (§5.1)	6	$O(n^2)$	$O(n^2)$	1	$f < n/3$	√	√
Wintermute (§5.2)	**2**	$O(n^2)$	$O(n^2)$	1	$f < n/3$	√	√
3Jane (§5.3)	**2**	$O(\sqrt{fn})$	$O(\sqrt{fn})$	$O(\sqrt{f/n})$	$f < n/5$	√	√

Table 1: Comparing BFT SMR protocols in a system of n processes with at most f failures. The metrics are: conflict-free latency (**L**), message complexity (**M**), authenticators complexity (**A**), load (**D**), and resilience (**R**). Leaderless protocols (√) achieve the same latency everywhere, while leader-driven protocols (×) need an additional message delay (+1) for non-leader processes. The commutativity of state-machine commands is either ignored (×), partially leveraged (√), or leveraged (√). The comparison is further detailed in §5.4.

permits to distribute the load and it avoids the potential disruptions due to a Byzantine leader. A common design is to rotate consensus instances [69]. Another approach is the use of a DAG data structure [7, 39, 65].

Leaderless protocols submit commands from multiple replicas at the same time. A key observation is that concurrent commands that commute need not to be ordered to reach linearizability [45]. Hence, to avoid ordering commands, some leaderless protocols [1, 5, 10, 30, 58] leverage commutativity at the application level. This reduces latency in good cases. Unfortunately, *none* of the existing leaderless solutions permits to execute a command after two message delays (see Table 1). EZBFT [5] claimed it was the first such protocol but it is actually flawed [62]. ISOS [30] 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 [58] 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, there is no BFT SMR protocol that is fast, i.e., which permits to execcute a conflict-free command from any replica in two message delays. This work intends to bridge such a gap. It presents protocols that execute conflict-free commands, that is commands which do not encounter concurrent non-commuting commands, after a single round-trip to a nearby quorum. In total, we present three Byzantine leaderless state-machine replication (BLSMR) protocols: *(i)* a Byzantine fault-tolerant variation of the Egalitarian Paxos protocol [54], *(ii)* a graph-based protocol named Wintermute, and *(iii)* a timestamp-based protocol called 3Jane. The protocols are written within a common framework. The framework introduces an abstract distributed machine to implement BLSMR. This simplifies exposition and underlines similarities between the solutions. As importantly, the machine also provides a principled approach to construct sound protocols—a difficult task as several algorithms were found incorrect [2, 61, 62, 67].

In a nutshell, the machine works as follows. It uses a directed graph to encode execution constraints, or *dependencies*, between non-commuting commands. To construct the graph, the

machine calls two services: a dependency discovery service (DDS) and a consensus service. Initially, a command is submitted at some replica who is in charge to coordinate its execution. For this, the replica first accesses the DDS service to discover its dependencies. Then, it calls consensus to commit them. A command execute once its dependencies are committed and executed at a replica. The machine also offers a fast path that skips consensus. Both Wintermute and 3Jane implement it using $n \geq 5f+1$ replicas. The fast path is taken when replicas report all the exact same conflicting commands. In the fast path, a command executes at its coordinator replica after a single round-trip to a nearby quorum.

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

The algorithms and abstractions presented in this work are specified in TLA$^+$. These specifications are available in an online companion repository [60].

2 Preliminaries

2.1 System model

We follow the standard model in [21]. The system consists of a set $\mathbb{P}$ of distributed processes, with $n = |\mathbb{P}|$. 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 [47].

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 to a correct process q, eventually q receives m (provided it executes enough receptions). Byzantines processes can spy on links connecting any two processes. The system is partially synchronous [29]: 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 bound and are unable to break the cryptographic techniques mentioned above.

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 ($\mathcal{C}$). 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 [73], we omit clients from most part of the discussion, and defer to the standard literature for issues regarding numbering, re-submission, and de-duplication of commands.

A service implemented with SMR is supposed to be *linearizable* [23, 38]. Informally, 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 [45, 57] 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. If c and c' do not commute, we say that they *conflict* and write $\mathsf{c} \asymp \mathsf{c}'$. 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.
- All local computations are instantaneous.

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

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.

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 synchronous failure-free run of the protocol is fast.

Notice that the above definition considers only failure-free runs. This comes from the fact that executing commands quickly would not make sense when Byzantine failures happen. As a faulty process may arbitrarily deviate from the protocol it can execute a command at any time.

2.3 Problem statement

We are interested with 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 [54] 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, 39, 52, 54, 58, 71, 74]. Below, we extend the work of Moraru, Andersen and Kaminsky [54] on Egalitarian Paxos to Byzantine systems.

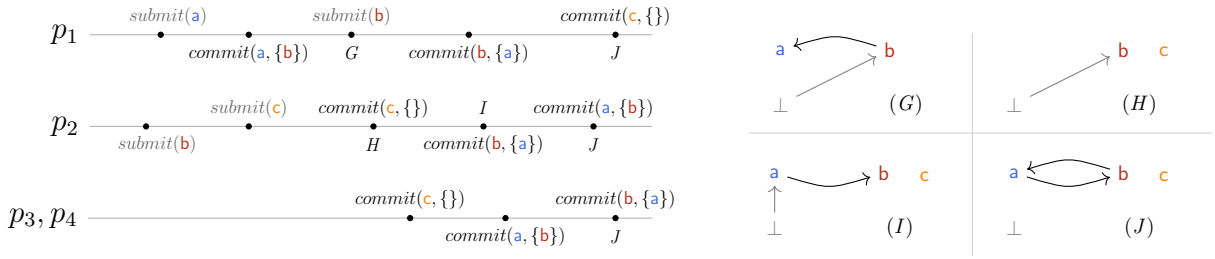


Figure 1: A run of BLSMR– *(left) processes submit and commit commands* a, b *and* c*; (right) the dependency graphs formed at the four distributed processes.*

3.1 Principles

A central notion in [54] is the *dependency graph*. This graph records how commands execute against the copy of the state machine. As observed in [45, 57], 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. Command b := T[Bob] ← 42$\boldsymbol{v}$ and c := T[Alice] ← 5$\boldsymbol{v}$ respectively assign 42$\boldsymbol{v}$ and 5$\boldsymbol{v}$ to the accounts of Alice and Bob (for some token $\boldsymbol{v}$). 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 a := T[Bob] that returns the balance of *Bob*.

In practice, commutativity is over-approximated with the notion of *conflicts*. We note ($\asymp$) the binary, non-reflexive, and symmetric relation over $\mathcal{C}$ that captures conflicts. In the above example, a single pair of commands conflict: a $\asymp$ 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 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, *phase*(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 *deps*(c) equals $\bot$. This happens upon calling *submit*(c) locally and when c is in the start phase. A command can be submitted by one or more processes. A call to *commit*(c, D) assigns $D \in 2^{\mathcal{C}}$ to *deps*(c). This requires the phase of c to be either start or pending. After this call, c reaches the commit phase. Let *deps**(c) be the transitive closure of the *deps* relation starting from {c}. Command c is stable once it is committed and so are its transitive dependencies in *deps**(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 *deps*(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, d $\in D$ or c $\in D'$.

Liveness: If a command is committed at some correct processes 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 commands a, while p_2 submits c. Both processes also submit comamnd 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 is similar to when dealing with crash failures in Egalitarian Paxos [54]. 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. 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, for instance using their identifiers.

To illustrate this mechanism, consider again Figure 1. Suppose that commands in a cycle are ordered using a lexicographical order, the smallest commands executing first. A possible execution order for the four 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. Notice that 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 in Figure 1.

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 state-machine replication protocol.

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 by Byzantine processes. Below, we introduce an operator to lighten the description of the algorithms that do such a verification.

Base notions. For starters, we model stateless verifications. 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 provable values (x, y), a function ϕ is *verifiable* if there exists a boolean function check_ϕ, called *the verifier of* ϕ, such that $\mathit{check}_\phi(x, y)$ evaluates to *true* if and only if $\phi(x) = y$. Given a verifiable function ϕ and two provable values x and y, we define a trusted assignment operator. This operator provides syntactic glue to check if the values are mapped with ϕ before doing the assignment. When this is not the case, an error is raised. Formally,

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

To illustrate, consider a verifiable random functions (VRF) [11]. Recall that given some pair of keys $(\mathrm{SK}, \mathrm{VK})$, where SK is a secret key and VK the corresponding public verification key, a VRF provides jointly *(i)* an evaluator $Eval_{\mathrm{SK}}(x)$ which given the key SK and some input x, returns an output y not distinguishable from a pseudo-random value, *(ii)* a prover $P_{\mathrm{SK}}(x)$ that takes as parameter SK and outputs a proof π that y matches the input x and VK, and *(iii)* a verifier $V_{\mathrm{VK}}(\pi, x, y)$, which verifies the proof. We map respectively $Eval_{\mathrm{SK}}(x)$ and $V_{\mathrm{VK}}(\pi, x, y)$ to the notions of verifiable function and verifier as defined above. For the verifier, the proof π is attached to both x and y to make them provable, i.e., $x.\Gamma = y.\Gamma = \pi$.

The above notion of trust is not always sufficient in a distributed system. This comes from the fact that distributed services can be stateful. In this situation, trust should account for the whole history of the service. Below, we extend the operator for such a purpose.

Definitions. In this work, we consider that a distributed service is a shared object offering a single operation [20]. For instance, consensus is modeled as an object offering the operation *propose* at its interface with the usual guarantees [55].

Given a run r and a service S, we write $r|S$ the events related to S in r. $S_r(x) = y$ holds when $r|S$ is a run of S and in this run the service S is called with x and returns y. When referring to the run where the evaluation takes place, the subscript r is simply omitted. Service S is *verifiable* when there exists a boolean function $check_S(x, y)$, for x and y provable, called the verifier of S, such that $check_S(x, y)$ holds iff $S(x) = y$ is true. We extend naturally the trusted assignment operator to work with services. In detail, for S verifiable,

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

We now 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. Consider x a provable value and T a verifiable service. 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 *trusted wrt.* (T, x).

Examples. Many practical systems abide by the above definitions. For instance in a PKI architecture [31], or when enclaves are available [24], 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 [14, 21]. Function $check(x, y)$ returns *true* when there exist a view v and a set of $2f + 1$ signed commit messages for y in v. PBFT can also verify that input x is valid before it takes any step. In this case, denoting $1_{\mathcal{C}}$ the service that indicates if x is a command, i.e., the indicator function of $\mathcal{C}$, PBFT is trusted wrt. $(1_{\mathcal{C}}, x)$.

4.2 The machine

The abstract machine to implement BLSMR appears 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, 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 all the concurrent (or prior) commands conflicting with c. 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 there are no concurrent conflicting commands.

Internals. Besides a dependency graph, Algorithm 1 uses three variables. First, for each command c, $\mathsf{CONS}_{\mathsf{c}}$ is an instance of consensus associated with c. As common, this service offers

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: (D, b) ←DDS,c— DDS.announce(c)
10:         if b = false then D ←CONS_c,D— CONS_c.propose(D)
11:         send(c, D, b) to all
12:         deps(c) ← ⊥

13: commit(c) :=
14:     pre: recv(c, D, b)
15:          phase(c) ∈ {start, pending}
16:     eff: deps(c) ←v,(c,D,b)— D
```

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 10 in Algorithm 1). Furthermore, consider that $announce_p(\mathsf{c})$ and $announce_q(\mathsf{c}')$ return respectively (D, b) and (D', b') at two correct processes p and q. 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. Weak Agreement enforces that commands announced in the fast path are returned elsewhere. That is, if a process p retrieves $(D, true)$ when announcing command c, any process that announces c must also retrieve D. Such a property is similar to the agreement property of adopt-commit [35], a base building block of consensus. The last property (Liveness) of DDS ensures 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. This abstraction is best effort in the sense that always returning all the processes is a valid implementation. If the coordinator of some command c is suspected (line 7), the local process steps in to commit c; this is called a *recovery*.

Both the dependency discovery and consensus services are trusted: DDS is trusted wrt. $(1_{\mathcal{C}}, \_)$, while $cons_{\mathsf{c}}$ 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 does a recovery (line 7). First, the command is announced to the system (line 9). The announcement generates a set of dependencies and a flag to indicate if the fast path is available (line 9). If not, the command goes through the *slow path* and consensus is called (line 10). Once a decision is taken, the process broadcasts it to everybody in the system (line 11). When this message is received, a verifiable function v is called (line 16). 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. As DDS is trusted

wrt. $(1_{\mathcal{C}}, \_)$, c is necessarily a submitted command. Alternatively, the command have 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 have also the same guarantees as in the previous case: DDS is called with input c and c is a submitted command.

Example. To illustrate the BLSMR machine, let us consider again the run in Figure 1. Such a run may enroll 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. Contrarily to 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 then later by p_1. Such a situation might occur as follows: Internally, p_2 retrieves $(\{\}, \mathit{false})$ when announcing b. Then, the 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 appears in Appendix D.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 the Weak Agreement property of DDS. As a consequence, the BLSMR machine guarantees that processes agree eventually 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 implemenations of the DDS and CONS services. This is the purpose of the present section. In particular, we explains how to take the fast path when submitted commands are conflict-free (such as c in Figure 1). This property is key to ensure that the state-machine replication protocol is fast.

5.1 A first solution: à la EPaxos

First of all, we implement BLSMR in the spirit of Egalitarian Paxos (EPaxos) [54]. For clarity, we consider a variation of EPaxos which does not use sequence numbers, nor 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 was not already there, then it 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 [50]. 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 $\mathit{quorums}()$ returns all the possible quorums. Its verifier takes as input a set of processes P, returning *true* iff $P \in \mathit{quorums}()$.

To announce a command c, a process broadcasts it (line 5). When a process p receives c, p appends c to its log then 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 reply from everybody in a quorum, the announcer calculates the

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) :=
11:    pre: recv(c) from q
12:         check_{1_c}(c, true)
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:        ∧ ∀j ∈ Q, check_{reply,j}(c, D_j)
19:        ∧ D = ⋃_{j∈Q} D_j
20:        ∧ ¬ b
21:    return v
```

union of all the dependencies (line 7). Then, it generates and embeds the proof necessary to make the service verifiable (line 8). The proof contains enough evidences for other processes to verify that the announcement was legitimate. For each process in the quorum, the proof stores the dependencies it replied. Once the proof is computed, the announcer returns the tuple (D, false) at line 9. The hardcoded value *false* indicates that there is no fast path in this implementation.

Function $\mathit{check}(\mathsf{c}, (D, b))$, 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 value obtained by taking the union of such dependencies, and (line 20) no fast path is taken.

Variable BQS is a dissemination quorum system [50]. It ensures that any two quorums intersects over $f + 1$ processes (the definiton is recalled in Appendix B). As a consequence, at least one correct process reports all the conflicting commands it has seen so far. This guarantees the Visibility property of the DDS service. Building upon this observation, we can show that Algorithm 2 is correct.

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 $Q_{\mathsf{a}} = \{p_1, p_3, p_4\}$ replies successfully to p_1, while b is announced using $Q_{\mathsf{b}} = \{p_2, p_3, p_4\}$. 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_{a} and Q_{b} intersect over $f + 1 = 2$ processes, this is not a problem. By taking the union over all replies, p_1 computes $(D_{\mathsf{a}} = \{\mathsf{b}\}, \mathit{false})$ as the result of announcing a. As required, a sees the conflict with b.

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 [14], asking that processes participate to $\mathsf{CONS_c}$ only if $\mathit{check}(\mathsf{c}, (D, b))$ holds for the verifier in Algorithm 2. To make consensus verifiable, we then define a verifier $\mathit{check}_{\mathsf{CONS}}$. This function is implemented by using the commit messages generated by HotStuff: these messages form the proof, and function $\mathit{check}_{\mathsf{CONS}}$ 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 ($\mathsf{CONS_c}$) exists per command c. For performance, the leader of the first view in $\mathsf{CONS_c}$ is set to $\mathit{coord}(\mathsf{c})$. In synchronous failure-free runs, this permits to commit command c at $\mathit{coord}(\mathsf{c})$ after four message delays.

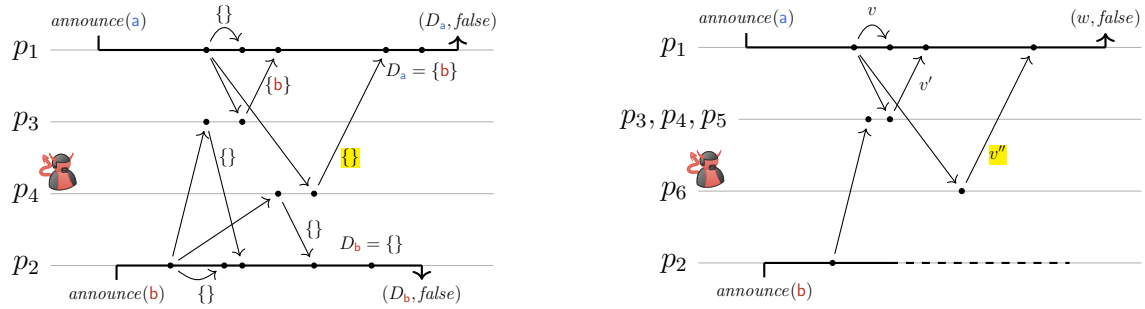


Figure 2: Fetching dependencies à la EPaxos (left) and Wintermute (right), $f = 1$.

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 [45, 68]. 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 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 [16]. 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.

To boost performance, we may further compact dependencies, as vectors of size $O(n)$. For this, computation is delegated to the coordinator. Unfortunately, if the coordinator is Byzantine it may assign twice the same sequence number. To remedy to this problem, sequence numbers are generated using a verifiable sequencing service, SEQ. 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 then s', 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 [9] implement a verifiable sequencing service. This costs one round-trip to a set of $4f + 1$ processes in charge of generating $\mathsf{c}.sn$. Such a request is executed in parallel with the announcement. Using this approach, dependencies are encoded as a compact representation (vector) for commands having verified sequence numbers, together with an explicit list.

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 [33]. 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: $\lfloor f \rfloor_{j \in Q} D_j = \bigcup \{\mathsf{c} : count(\mathsf{c}) \geq f + 1\}$, 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.

For instance, assume that the following dependencies are reported (0s are omitted): $v = \{p_1 \mapsto 1, p_2 \mapsto 2\}$, $v' = \{p_1 \mapsto 1, p_2 \mapsto 3\}$, and $v'' = \{p_2 \mapsto 99\}$. If process p_2 has not submitted more than 5 commands yet, a Byzantine node is trying to create an illegitimate dependency with

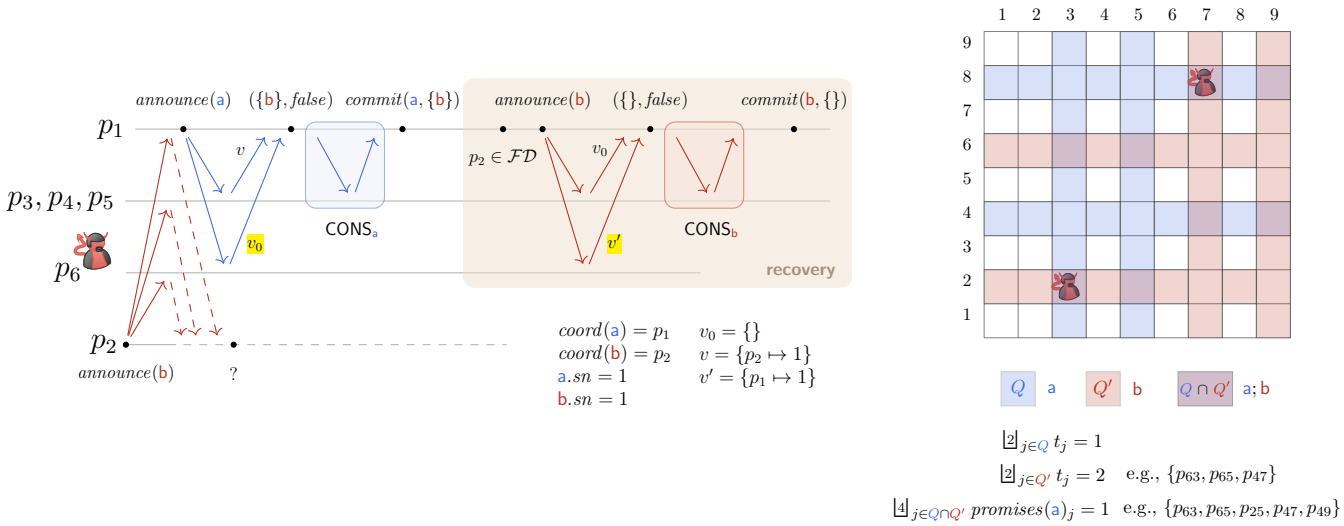


Figure 3: Illustration of the BLSMR protocols: *(left)* Wintermute with $n = 6$, $f = 1$; *(right)* 3Jane using a grid-like quorum system with $n = 81$, $f = 2$.

v''. Applying $\lfloor f \rfloor_{j \in Q} D_j$ for $f = 1$, the dependencies are $w = \{p_1 \mapsto 1, p_2 \mapsto 3\}$. The threshold union operator is shielding us from the attack. This situation is illustrated in Figure 2(right). In this figure, a and b are submitted respectively by p_1 and p_2, with a.$sn = 2$ and 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 respectively v and v''.

The threshold union operator only returns submitted commands. However, it might also remove legitimate dependencies. For instance, in the above example, $p_2 \mapsto 4$ is removed but it could be legitimate if p_2 had already submitted enough commands. To avoid this, we use large quorum intersections, namely of size $3f + 1$. We call such a quorum system a *witnessing quorum system* (WQS). (A formal definition can be found in Appendix B.)

In Figure 2(right), $f = 1$ and the two quorums are $Q_\mathsf{a} = \{p_1, p_3, \ldots, p_6\}$ and $Q_\mathsf{b} = \{p_2, \ldots, p_6\}$. Because of the intersection, either $f + 1$ processes in Q_a report b, or the converse holds for Q_b wrt. a. In Figure 2(right), the former case happens as v' is reported by the processes $\{p_3, p_4, p_5\}$. Large quorum intersections enforce the Visibility property of the DDS service.

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. This protocol applies the techniques introduced in the previous section. The differences with Algorithm 2 are highlighted in blue. Commands have sequence numbers generated by either clients or a sequencing service, allowing to compact dependencies. 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), as illustrated in Figure 2(right).

Fast Path. A key difference with Algorithm 2 is the fast path in line 9. The fast path is permitted when function *fast*() returns *true*. This happens provided that all processes answer (line 24) and that they report the exact same dependencies (line 25). The intuition is as follows: Consider that p retrieves $(D, true)$, while q obtain $(D', \_)$ when announcing command c. Because p can take the fast path, a command $\mathsf{d} \in D$ is reported n times to p. Hence, in the quorum used by q it is reported $f + 1$ times, and 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

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(),
               ∀j ∈ Q, ∃D_j ⊆ C, recv(c, D_j) from j
7:         D ← ⌊f⌋_{j∈Q} D_j
8:         D.Γ ← (j, D_j)_{j∈Q}
9:         return (D, fast(c, D))

10: reply(c) :=
11:     pre: recv(c) from q
12:         check_{1_C}(c, true)
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:         ∧ ∀j ∈ Q, check_{j,reply}(c, D_j)
19:         ∧ D = ⌊f⌋_{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
```

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. Thanks to this argumentation, we can establish that Algorithm 3 is correct.

Proposition 2. Algorithm 3 implements a trusted DDS service.

Consensus. As with Algorithm 2, Wintermute may use any consensus protocol for the CONS service. In the common case, only the coordinator of command c proposes a value to $\mathsf{CONS}_{\mathsf{c}}$. 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 recent solution is the one proposed in [42]. 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.

Example. Figure 3(left) depicts a run of Wintermute. In this run, process p_1 and p_2 submit respectively a and b. Command a is announced using quorum $Q = \{p_1, p_3, \ldots, p_6\}$. Because p_6 reports $v_0 = \{\}$, omitting b, the fast path cannot be taken. Hence a goes on the slow path using consensus $\mathsf{CONS}_{\mathsf{a}}$. The consensus proposal of p_1 is $\lfloor f \rfloor_{j \in Q} D_j = v$, which encodes {b}. This value is decided and command a committed. Concurrently, process p_2 announces command b to $Q' = \{p_2, \ldots, p_6\}$. Before any decision is taken, p_2 gets slow. Because b is a dependency of a, process p_1 starts recovering command b. It uses quorum $\{p_1, p_3, \ldots, p_6\}$ to annonuce b. $\lfloor f \rfloor_{j \in Q} D_j$ returns $\{\}$ despite that p_6 is reporting maliciously $v' = \{p_1 \mapsto 1\}$ that encodes {a}. This matches the dependencies that p_2 may have computed on the fast path. Process p_1 proposes such dependencies to $\mathsf{CONS}_{\mathsf{b}}$ which are decided.

Performance. Consider a conflict-free command c submitted to Wintermute during a synchronous failure-free run. Its coordinator announces c using Algorithm 3. Because c is conflict-free, there are no concurrent commands conflicting with it. That is, commands conflicting with c and submitted previously are already executed everywhere at the time c is submitted. In Algorithm 3, the processes compute the same dependencies for c, triggering the fast path. This path skips consensus in Algorithm 1, committing the command at *coord*(c). Command c becomes stable and it gets executed immediately (see §3.2). From what precedes, c executes after a single round-trip. This is a well-known lower bound [44], and to the best of our knowledge Wintermute is the first protocol to reach it.

Theorem 3. Wintermute is a fast state-machine replication protocol.

5.3 Third solution: 3Jane

Blockchain can span hundreds to thousands of machines. At this scale, it is preferable that not all the processes participate to the ordering of every command. To this end, we present a protocol reducing load, that is the worst-case probability a process gets chosen to order a command. This third solution, called 3Jane, is specifically tailored for the case $f \ll n$.

Improving scalability. 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 executes CONS only at a subset of machines (e.g., just $3f + 1$). The following observation motivates then a second refinement: in BLSMR, CONS_c and CONS_d *may run in parallel.* We leverage this to spread the load: the processes running CONS_c are chosen with a verifiable random function that takes as input c, in the manner of Algorand [36], Explicit dependencies in Leaderless SMR degrade tail latency [68], a phenomena more likely to happen at scale. Thus, our third refinement replaces explicit dependencies with timestamps. The 3Jane protocol implements all these refinements. For clarity, we first present a base version of the protocol, to which we add successively a fast path and a mechanism to leverage commutativity.

Protocol design. The DDS service of 3Jane appears in Algorithm 4. The main differences with Algorithm 3 are highlighted in blue. Variable BQS (line 2) is now a masking quorum system, ensuring that any two quorums intersect over $2f + 1$ processes [50]. Algorithm 4 uses one more variable than Wintermute (line 4); its role will be clarified shortly.

As previously, p announces a command c by contacting all the processes and awaiting for a quorum to answer. Upon receiving the announcement, a process q appends c to its log, then it replies to p the position of c in the log (line 16). When doing so, process q *promises* to not accept commands at a position lower than the one occupied by c in the log. Process p gathers all such replies to compute a timestamp for c. Timestamping faces the very same problem as the one related to the compaction of dependencies: A Byzantine process may lie about the position of c in its log. To deal with such a concern, a timestamp t is legitimate only if $f + 1$ processes replied a timestamp $t' \geq t$. In line 8, process p assigns the largest legitimate timestamps to c. For this, p uses a generalized version of the threshold union operator (see Appendix C for a formal definition).

Algorithm 4 returns a timestamp t at line 11. Dependencies are inferred from timestamp t using variable *promises*. (Ties are broken using a canonical total order $<$ on the commands, e.g., lexicographically.) Variable *promises* is updated with the proof in $t.\Gamma$ when command c is announced (lines 17 to 21). In detail, for some process j, $promises[j][\mathsf{d}]$ stores the promise of j about command d. Initially, $promises[j][\mathsf{d}]$ is set to 0. Function $promises(\mathsf{d})_j$ is defined as $promises[j][\mathsf{d}]$ if this value is defined and $max_{\mathsf{d} \in \mathcal{C}} \{promises[j][\mathsf{d}]\} + 1$ otherwise. The dependencies of c are all the commands d such that for some quorum Q in $BQS.quorums()$, $2f$ $_{j \in Q}$ $promises(\mathsf{d})_j$ is lower than t. The rationale behind this computation is as follows: Algorithm 4 excludes d from the dependencies of c when every possible annonucement of d returns a higher timestamp than t. Because there are at most f Byzantine failures, this happens for sure when in each quorum $f+1$ correct processes report a higher timestamp than t for the command. Hence, the threshold union operator is applied to the subsets of size $2f + 1$ in each quorum. This observation is key to establish the correctness of Algorithm 4.

Proposition 3. Algorithm 4 implements a trusted DDS service.

Example. 3Jane is illustrated in Figure 3(right). In this figure, variable BQS is a (grid-like) witnessing Byzantine quorum system. 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 spread the load: each process has around 39% of chance to participate in the ordering of a command, against 69% 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 receives first command a then b. There are two Byzantine processes, p_{23}

Algorithm 4 3Jane: timestamping commands – code at process p

```
1: Variables:
2:     BQS                                              // masking quorum system
3:     L
4:     promises                                         // bi-dimensional array (local to p)

5: announce(c) :=
6:     eff: send(c) to all
7:          wait until ∃Q ∈ BQS.quorums(),
                ∀j ∈ Q, recv(c, t_j) from j
8:          t ← ⌊f⌋_{j∈Q} t_j
9:          t.Γ ← (j, t_j)_{j∈Q}
10:         send(c, t) to all
11:         return (t, false)

12: reply(c) :=
13:    pre: recv(c) from q
14:         check_{1_c}(c, true)
15:    eff: t ← L.append(c)
16:         send(c, t) to q

17: update_promise(c, j) :=
18:    pre: recv(c, t) from _
19:         (j, t_j) ∈ t.Γ ∧ check_{j,reply}(c, t_j)
20:         t_j = 1 + max_{d∈C} {promises[j][d]}
21:    eff: promises[j][c] ← t_j

22: check(c, (t, b)) :=
23:    let (j, t_j)_{j∈Q} = t.Γ
24:    v ← ∧ check_BQS(Q)
25:        ∧ ∀j ∈ Q, check_{j,reply}(c, t_j)
26:        ∧ t = ⌊f⌋_{j∈Q} t_j
27:        ∧ ¬ b
28:    return v
```

and p_{87}. If all the processes in Q respond honestly, a takes the fast path: they all report $\lfloor 2 \rfloor_{j \in Q}\, t_j = 1$. This timestamp is comuputed when recovering a, as $\lfloor 2 \rfloor_{j \in Q'}\, t_j$ also equals 1 for any $Q' \subseteq Q$ with $|Q'| = |Q| - 2$. Similarly, b is announced with timestamp 2 when quorum Q' replies. Command a is a dependency because $\lfloor 4 \rfloor_{j \in Q \cap Q'}\, promises(\mathsf{a})_j = 1$. For instance, the correct processes $\{p_{63}, p_{65}, p_{25}, p_{47}, p_{49}\}$ have all promised timestamp 1 to a.

Fast path. As with Wintermute, we now add a fast path to skip consensus and commit conflict-free commands in a single round-trip. To this end, we follow the approach in Tempo [32], with a twist to accommodate Byzantine failures. It works as follows: BQS now returns witnessing Byzantine quorums. Each command c has a fixed quorum computed by calling function $BQS.getQuorum(\mathsf{c})$. The quorum assigned to c is verifiable using BQS. Only the coordinator of c can take the fast path. To do so, it contacts exactly the processes in Q at line 6. The fast path is taken when every process in quorum Q answers $\lfloor f \rfloor_{j \in Q}\, t_j$. In which case, the flag at line 11 is set to *true*. When p recovers c, it also contacts quorum $Q = BQS.getQuorum(\mathsf{c})$. However, instead of waiting for the quorum in full at line 7, it awaits $|Q| - f$ replies. The computation continues as in Algorithm 4 with such replies. The verifier is adjusted appropriately to verify the fast path.

The additional processes in the fast-path quorum provide redundancy to retrieve the result of the fast path: for any $P \subseteq Q$ of $|Q| - f$ processes, the threshold union over P equals the one over Q when the fast path is taken. This explains the correctness of such a variation.

Proposition 4. Algorithm 4 with a fast path is trusted DDS service.

When Algorithm 4 is used in Algorithm 1, the proof is already disseminated in the commit message. Thus, the computation in line 10 is not necessary. As a consequence, only the processes in Q participates in the fast-path variation of 3Jane. This reduces load from 1 to $O(\sqrt{f/n})$, that is the load of the witnessing quorum system (see Appendix B). The fast path is taken when all the processes in Q are correct and enough of them report the same timestamp. Since Byzantine attacks are rare, it is likely that the first part of this condition holds. We may increase the chance that the second part also holds by leveraging commutativity to compute timestamps, as detailed below.

Commutativity. The timestamping mechanism can be adjusted to leverage the commutativity of state-machine commands. A base case is when the conflict relation is split into equivalent classes. In this situation, we use a dedicated log per conflict class. Then, to take into account read/write conflicts in each class, e.g., if the service is a key-value store, we return a pair

(k, b) instead of the position of the command in the log, where k is an integer and b a boolean. In detail, if c is a write added to the log at (say) position k, the process replies $(k, 0)$; otherwise when c is a read, the process skips adding it to the log and returns $(k, 1)$ where k is the highest position occupied in the log. Pairs compare as tuples, i.e., $(k, b) > (k', b') \triangleq k > k' \vee (k = k' \wedge b > b')$ when used in Algorithm 4. Under the assumption that commutativity in the state machine is accurately modeled in this manner, we can show the following result.

Theorem 4. 3Jane is a fast state-machine replication protocol.

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. The table also indicates protocols that leverage *commutativity*, that is protocols that order only non-commuting (conflicting) state-machine commands. Table 1 draws a comparison in the common case, that is when the system is synchronous and failure-free. Clients are local to the state-machine processes (proxy model). Measures are taken in the critical path, from the moment a command is submitted to when it is executed at the local replica. Table 1 lists the following metrics of interest:

- **L** (conflict-free latency): Worst-case number of message delays in the critical path for a conflict-free command. A command is *conflict-free* when at the time it is submitted, all the commands already submitted and conflicting with it are executed everywhere.
- **M** (message complexity): Total number of messages exchanged in the critical path.
- **A** (authenticator complexity): Total number of signature verifications in the critical path.
- **D** (load): Worst-case probability of a process to be part of the critical path. This is analogous to the load for quorum systems [51].
- **R** (resilience): Maximum number of failures a protocol tolerates.

Takeaways. Wintermute and 3Jane execute a conflict-free command at its local replica after a single round-trip. For commands having no return value, such as blind writes or transactions in blockchain, this is sufficient for the client to return: the proof that permitted the command to execute locally contains enough information. When the Byzantine adversary is weak ($f \ll n$), the load of 3Jane is optimal [51] and satisfies that: $lim_{n \to \infty}\ \mathbf{L} = 0$. This comes from the fact that consensus runs exactly over $O(f)$ processes, and that this set of processes 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.

6 Related work

Protocols. PBFT [21] provides the first practical partially-synchronous BFT SMR solution. This solution is similar to Paxos [46] for the crash-failure case, requiring additional message exchanges to verify the computation performed at the leader. This additional work is known as the Dolev-Reischuk lower bound [28].

Zyzzyva [40] 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, 37] 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 return right away. Otherwise, the client generates an abort history and the command goes through a slow path implemented with PBFT.

In [39], 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., [13, 17]). 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 [26]. The work in [65] adds a fast-path optimization and further reduces costs.

The use of commutativity in SMR is introduced in [45, 57]. PGB [58] adapts Generic Broadcast [57] 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 [9]. It requires $n \geq 4f+1$ processes and has $O(n^2)$ message complexity. It bears similarity with 3Jane, using timestamps to order commands. However the protocol only leverages commutativity at execution time. Ordering still requires a slow path that necessitates to always run an agreement protocol.

In some 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 [58] protocols. Basil [66] is a also recent work that rely 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. Optimistic execution can be added to the protocols in §5. However, it is not always applicable because it necessitates that side-effects can rollback.

We can deconstruct existing leaderless solutions (e.g., the ones in Table 1) with the help of the abstract machine depicted in §4. In the vein of prior works on Paxos [12], 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 to compare solutions for the fail-stop and Byzantine failures 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 [21] and HotStuff [73] manipulate *certificates*, or *quorum certificates*, that is authenticated collections of messages that permit to verify computation. The seminal work in [18] 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 [18], the authors introduce the notion of external validity for Byzantine agreement. The work in [22] studies how validity impacts agreement. Validity bound the effects of Byzantine processes on a computation and permits to connect 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 [64]. Inline with these works, the notions introduced in §4.1 offer a convenient framework to describe trust in a distributed computation.

Decomposition. Several works [49, 59, 72] 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 [43], hardening replication protocols to tolerate Byzantine faults is challenging. Some prior leaderless protocols (e.g., ezBFT [5]) are known for bugs. This shows that achieving such a generalization is not straightforward.

In [26], the authors propose Narwhal, a mempool abstraction which bears similarities with 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 [63] proposes a leaderless protocol aimed at implementing a digital currency where consensus is only used

[1] Table 1 reports on the performance of DAG-Rider when combined with [13]. In [39], the authors consider amortized message complexity and the impossibility of man-in-the-middle attacks. Under such assumptions, μ and α reduce respectively to $O(n)$ and $O(1)$.

for transaction conflict resolution whereas transaction dissemination is done via a broadcast protocol. The resulting protocol has 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 [45], 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 permits to compose 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.

7 Conclusion

This work presents an abstract state machine to construct leaderless BFT SMR protocols in a principled manner. Three efficient implementations of the machine are detailed: an approach à la Egalitarian Paxos, a graph-based solution, and a timestamp-based one. The protocols all leverage the commutativity of state-machine commands for improved performance. Two of these solutions can execute conflict-free commands in optimal time, after just one round-trip to a nearby quorum.

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. Association for Computing Machinery. ISBN 1595930795. doi: 10.1145/1095810.1095817. URL https://doi.org/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. abs/1712.01367. URL http://arxiv.org/abs/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. doi: 10.4230/LIPIcs.OPODIS.2018.16. URL https://doi.org/10.4230/LIPIcs.OPODIS.2018.16.

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18. Association for Computing Machinery. ISBN 9781450355841. doi: 10.1145/3190508.3190538. URL https://doi.org/10.1145/3190508.3190538.

[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. doi: 10.1109/ICDCS.2019.00063. URL https://doi.org/10.1109/ICDCS.2019.00063.

[6] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. 32(4). ISSN 0734-2071. doi: 10.1145/2658994. URL https://doi.org/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. 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. Association for Computing Machinery. ISBN 9781450367325. doi: 10.1145/3318041.3355458. URL https://doi.org/10.1145/3318041.3355458.

[9] Rida A. Bazzi and Yin Ding. Non-skipping timestamps for byzantine data storage systems. In Rachid Guerraoui, editor, *Distributed Computing*, pages 405–419. Springer Berlin Heidelberg. ISBN 978-3-540-30186-8.

[10] Rida A. Bazzi and Maurice Herlihy. Clairvoyant state machine replication. abs/1905.11607. URL http://arxiv.org/abs/1905.11607.

[11] Nir Bitansky. Verifiable random functions from non-interactive witness-indistinguishable proofs. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography*, pages 567–594. Springer International Publishing. ISBN 978-3-319-70503-3.

[12] Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. Deconstructing paxos. 34(1):47–67. doi: 10.1145/637437.637447. URL https://doi.org/10.1145/637437.637447.

[13] Gabriel Bracha. Asynchronous byzantine agreement protocols. 75(2):130–143. ISSN 0890-5401. doi: https://doi.org/10.1016/0890-5401(87)90054-X. URL https://www.sciencedirect.com/science/article/pii/089054018790054X.

[14] Manuel Bravo, Gregory V. Chockler, and Alexey Gotsman. Making byzantine consensus live. 35(6):503–532. doi: 10.1007/s00446-022-00432-y. URL https://doi.org/10.1007/s00446-022-00432-y.

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

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

[17] 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. doi: 10.1007/11561927_42. URL https://doi.org/10.1007/11561927_42.

[18] 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. Springer-Verlag. ISBN 3540424563.

[19] 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. doi: 10.1145/3519270.3538430. URL https://doi.org/10.1145/3519270.3538430.

[20] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Unifying concurrent objects and distributed tasks: Interval-linearizability. 65(6). ISSN 0004-5411. doi: 10.1145/3266457. URL https://doi.org/10.1145/3266457.

[21] 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. USENIX Association. ISBN 1880446391.

[22] 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. Association for Computing Machinery. ISBN 9798400701214. doi: 10.1145/3583668.3594567. URL https://doi.org/10.1145/3583668.3594567.

[23] 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. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-210-5. doi: 10.4230/LIPIcs.DISC.2021.18. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.DISC.2021.18.

[24] Victor Costan and Srinivas Devadas. Intel SGX explained. page 86. URL http://eprint.iacr.org/2016/086.

[25] Tyler Crain, Christopher Natoli, and Vincent Gramoli. Evaluating the red belly blockchain. abs/1812.11747. URL http://arxiv.org/abs/1812.11747.

[26] 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. Association for Computing Machinery. ISBN 9781450391627. doi: 10.1145/3492321.3519594. URL https://doi.org/10.1145/3492321.3519594.

[27] Volker Diekert and Grzegorz Rozenberg, editors. *The Book of Traces*. World Scientific. ISBN 978-981-02-2058-7. doi: 10.1142/2563. URL https://doi.org/10.1142/2563.

[28] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. 32(1):191–204. ISSN 0004-5411. doi: 10.1145/2455.214112. URL https://doi.org/10.1145/2455.214112.

[29] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. 35(2):288–323. ISSN 0004-5411. doi: 10.1145/42282.42283. URL https://doi.org/10.1145/42282.42283.

[30] 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. doi: 10.1109/PRDC53464.2021.00019. URL https://doi.org/10.1109/PRDC53464.2021.00019.

[31] James H Ellis. The possibility of secure non-secret digital encryption. 8.

[32] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. Efficient replication via timestamp stability. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 178–193. ACM, . doi: 10.1145/3447786.3456236. URL https://doi.org/10.1145/3447786.3456236.

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

[34] Matthias Fitzi, Aggelos Kiayias, Giorgos Panagiotakos, and Alexander Russell. Ofelimos: Combinatorial optimization via proof-of-useful-work - A provably secure blockchain protocol. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part II*, volume 13508 of *Lecture Notes in Computer Science*, pages 339–369. Springer. doi: 10.1007/978-3-031-15979-4_12. URL https://doi.org/10.1007/978-3-031-15979-4_12.

[35] Eli Gafni. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, PODC '98, pages 143–152. ACM. ISBN 0-89791-977-7.

[36] 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. doi: 10.1145/3132747.3132757. URL https://doi.org/10.1145/3132747.3132757.

[37] 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. doi: 10.1109/DSN.2019.00063.

[38] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. 12(3):463–492. ISSN 0164-0925. doi: 10.1145/78969.78972. URL https://doi.org/10.1145/78969.78972.

[39] 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. Association for Computing Machinery. ISBN 9781450385480. doi: 10.1145/3465084.3467905. URL https://doi.org/10.1145/3465084.3467905.

[40] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. 27(4). ISSN 0734-2071. doi: 10.1145/1658357.1658358. URL https://doi.org/10.1145/1658357.1658358.

[41] Klaus Kursawe. Optimistic byzantine agreement. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, SRDS '02, page 352. IEEE Computer Society. ISBN 0769516599.

[42] 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. Association for Computing Machinery. ISBN 9781450385480. doi: 10.1145/3465084.3467924. URL https://doi.org/10.1145/3465084.3467924.

[43] 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, . doi: 10.1007/978-3-642-24100-0_22. URL https://doi.org/10.1007/978-3-642-24100-0_22.

[44] Leslie Lamport. Lower bounds for asynchronous consensus. 19(2):104–125, .

[45] Leslie Lamport. Generalized consensus and Paxos, .

[46] Leslie Lamport. The part-time parliament. 16(2):133–169, . ISSN 0734-2071. doi: 10.1145/279227.279229. URL https://doi.org/10.1145/279227.279229.

[47] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. 4 (3):382–401. ISSN 0164-0925. doi: 10.1145/357172.357176. URL https://doi.org/10.1145/357172.357176.

[48] Giuliano Losa, Eli Gafni, and David Mazières. Stellar consensus by instantiation. In Jukka Suomela, editor, *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, volume 146 of *LIPIcs*, pages 27:1–27:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, . doi: 10.4230/LIPIcs.DISC.2019.27. URL https://doi.org/10.4230/LIPIcs.DISC.2019.27.

[49] 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, . doi: 10.1145/2933057.2933072. URL https://doi.org/10.1145/2933057.2933072.

[50] Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. 11(4):203–213. doi: 10.1007/s004460050050. URL https://doi.org/10.1007/s004460050050.

[51] Dahlia Malkhi, Michael K. Reiter, and Avishai Wool. The load and availability of byzantine quorum systems. 29(6):1889–1906. doi: 10.1137/S0097539797325235. URL https://doi.org/10.1137/S0097539797325235.

[52] 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.

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

[54] 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.

[55] Achour Mostéfaoui and Michel Raynal. Signature-free asynchronous byzantine systems: from multivalued to binary consensus with t¡ n/3, o(n^2) messages, and constant time. 54(5):501–520. doi: 10.1007/s00236-016-0269-y. URL https://doi.org/10.1007/s00236-016-0269-y.

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

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

[58] 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. doi: 10.1007/978-3-642-25873-2_23. URL https://doi.org/10.1007/978-3-642-25873-2_23.

[59] 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, . doi: 10.4230/LIPIcs.DISC.2020.24. URL https://doi.org/10.4230/LIPIcs.DISC.2020.24.

[60] Tuanir França Rezende and Pierre Sutra, . URL https://github.com/otrack/bft-leaderless-gen-ai/.

[61] Nibesh Shrestha and Mohan Kumar. Revisiting hbft: Speculative byzantine fault tolerance with minimum cost. abs/1902.08505, . URL http://arxiv.org/abs/1902.08505.

[62] Nibesh Shrestha and Mohan Kumar. Revisiting EZBFT: A decentralized byzantine fault tolerant protocol with speculation. abs/1909.03990, . URL http://arxiv.org/abs/1909.03990.

[63] 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, pages 299–313. Springer. ISBN 978-3-031-21016-7. doi: 10.1007/978-3-031-21017-4_20. 24th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2022); Conference Location: Clermont-Ferrand, France; Conference Date: November 15-17, 2022.

[64] 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. IEEE Computer Society. ISBN 9780769546711. doi: 10.1109/EDCC.2012.32. URL https://doi.org/10.1109/EDCC.2012.32.

[65] 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. Association for Computing Machinery. ISBN 9781450394505. doi: 10.1145/3548606.3559361. URL https://doi.org/10.1145/3548606.3559361.

[66] 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. Association for Computing Machinery. ISBN 9781450387095. doi: 10.1145/3477132.3483552. URL https://doi.org/10.1145/3477132.3483552.

[67] Pierre Sutra. On the correctness of egalitarian paxos. 156:105901. doi: 10.1016/J.IPL.2019.105901. URL https://doi.org/10.1016/j.ipl.2019.105901.

[68] 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. URL https://www.usenix.org/conference/nsdi21/presentation/tollman.

[69] 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. IEEE Computer Society. ISBN 9780769538266. doi: 10.1109/SRDS.2009.36. URL https://doi.org/10.1109/SRDS.2009.36.

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

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

[72] Michael J. Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M. Hellerstein, and Ion Stoica. Sok: A generalized multi-leader state machine replication tutorial. 1(1), . doi: 10.5070/sr31154817. URL https://doi.org/10.5070/sr31154817.

[73] 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. Association for Computing Machinery. ISBN 9781450362177. doi: 10.1145/3293611.3331591. URL https://doi.org/10.1145/3293611.3331591.

[74] Piotr Zieliński. Optimistic generic broadcast. In Pierre Fraigniaud, editor, *Distributed Computing*, pages 369–383. Springer Berlin Heidelberg. ISBN 978-3-540-32075-3.

A Linearizable objects with BLSMR

This section establishes that BLSMR implements a state-machine replication protocol. Our reasoning and algorithms are based upon the notion of trace [27]. We first introduce some preliminary concepts then detail the construction.

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 $.\mathsf{st}$ and $.\mathsf{val}$ selectors to respectively extract the state and the response value, i.e., given a state s and a command c, we let $\tau(s, c) = (\tau(s, c).\mathsf{st}, \tau(s, c).\mathsf{val})$. Without lack of generality, we consider that commands are applicable to every state.

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 occurence 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:

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

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

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

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

Lemma 2 ([27]). *$x \sim y$ holds iff $\mathsf{cmd}(x) = \mathsf{cmd}(y)$ and for any $c \not\parallel d$, $c^i <_x d^j \Leftrightarrow c^i <_y d^j$.*

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

Command traces. A trace captures a class of words that are equivalent, that is they sort non-commuting commands in the same order. It can seen as as special case of the notion of c-struct used to define the generalized consensus problem [45]. 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 [27].

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 $c \not\parallel d$, $c^i <_y d^j \wedge d^j \in x \Rightarrow c^i <_x d^j$, 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 $c \not\parallel d$ 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.

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

Algorithm 5 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)
```

– 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* [38] when it can be extended (by appending zero or more responses) into some history h' such that $complete(h')$ is equivalent to a legal and sequential history l preserving the real-time order in h, i.e., $\prec_{h'} \subseteq \prec_l$.

A.2 Algorithm

Algorithm 5 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 process also takes care of returning the result of the command to the caller.

In more detail, Algorithm 5 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 5, 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 [73], clients are correct.

At a process, Algorithm 5 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 ($\nparallel$) among commands; that is for any two commands c and d, if $\mathsf{c} \nparallel \mathsf{d}$ 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 5 is atomic. To apply a command c on the shared object, a client process executes $invoke(\mathsf{c})$ 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) $\mathsf{c} \in \mathit{deps}^*(\mathsf{d})$ and $\mathsf{d} \notin \mathit{deps}^*(\mathsf{c})$; or *(ii)* $\mathsf{c} \in \mathit{deps}^*(\mathsf{d})$, $\mathsf{d} \in \mathit{deps}^*(\mathsf{c})$ and $\mathsf{c} < \mathsf{d}$. Relation $\rightarrow$ defines the *execution order* at a replica process. If there is a one-way transitive dependency between two commands, Algorithm 5 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 5. 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* $p \in \mathit{Correct}$ *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 $\mathit{execute}(\mathsf{d})$ or $\mathit{execute}(\mathsf{c})$ takes place at p. In the first case, according to the way a batch is built at line 16, $\mathsf{c} \in \mathit{deps}^*(\mathsf{d})$ holds. Because c executes first, line 17 requires that $c \rightarrow d$ is true. In the second case, applying the same argument, $\mathsf{d} \in \mathit{deps}^*(\mathsf{c})$ 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 comamnds 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: $\mapsto = \cup \bigcup_{p \in \mathit{Correct}} \mapsto_p$.

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 $\mathit{deps}(\mathsf{c})$ and $\mathit{deps}(\mathsf{d})$ 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 lack of generality, assume that $\mathsf{d} \in D$ holds. We know that $\mathsf{d} \in \mathit{deps}_p^*(\mathsf{c})$ 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, $\mathsf{c} \rightarrow \mathsf{d}$. As a consequence, $\mathsf{c} \in \mathit{deps}_p^*(\mathsf{d})$ 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, $\mathsf{c} \in \mathit{deps}_q^*(\mathsf{d})$ 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 state-machine replication protocol.

Proof. (**Validity**) Assume that c gets executed at line 18 in Algorithm 5. 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 $\mathit{deps}^*(\mathsf{c})$ 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 Liveness property of BLSMR, 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 pending command $\mathsf{c}_1 \in deps^*(\mathsf{c})$ at time t_1. Again, by the Liveness property, c_t is eventually committed at q at some time $t_2 > t_1$. By repeating the above reasoning, there are infinitely many pending commands $(\mathsf{c}_i)_{i\geq 1}$ in the run. For each such commands 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 13) and response (line 18) 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 equals 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 ${S_p}' = \tau^*(S_p, L'_p).\mathsf{st}$, with $L'_p = L_p c$. By induction, $S_p = \tau^*(s_0, L_p).\mathsf{st}$. It follows that:

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

□

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

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, $Lc = L_p^r|_{\leq c}$. □

Next, we construct a linearization of h. Let E be the commands executed during r, that is $E = \bigcup_{p \in Correct} \mathsf{cmd}(L_p^r)$. 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, $\mathsf{c} \mapsto_p \mathsf{c}$. Hence, at the time it executes d, p has already executed d. □

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

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

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 p 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 5, the history h induced by r is linearizable.*

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

B Byzantine Quorum Systems

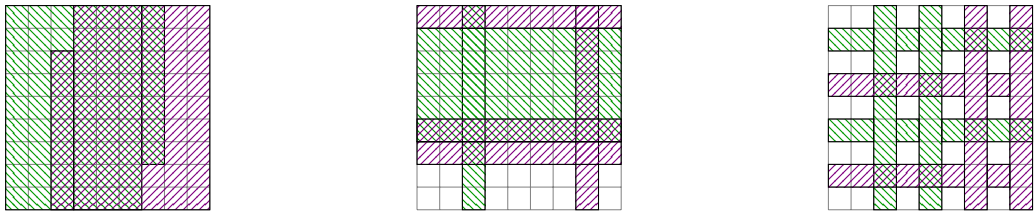
Byzantine quorum systems are standard mechanisms to hide faulty processes in BFT SMR. They extend quorum systems (from the crash-failure model) to deal with Byzantine failures. Below, we present this notion following standard definitions [50, 70].

A quorum system $\mathcal{Q} \subseteq 2^{\mathbb{P}}$ is a non-empty set of subsets of $\mathbb{P}$, every pair of which intersects. Each $Q \in \mathcal{Q}$ 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, $faulty(r) \in \mathcal{B}$). 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 $\mathcal{B} = \{B \subseteq \mathbb{P} : |B| = f\}$. A *Byzantine quorum system* (BQS) is a quorum system together with an adversary.

A base BQS is the *dissemination quorum system*, or DQS for short. Such a quorum system is defined through the following properties: (D-Consistency) $\forall Q_1, Q_2 \in \mathcal{Q},\ \forall B \in \mathcal{B},\ Q_1 \cap Q_2 \not\subseteq B$ (Availability) $\forall B \in \mathcal{B},\ \exists Q \in \mathcal{Q},\ B \cap \mathcal{Q} = \varnothing$ 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. A base example of such a quorum system are all the sets of $2f+1$ processes among $n = 3f+1$. Two quorums in this system when $n = 9^2$ and $f = 30$ are illustrated in Figure 4a.

A *masking quorum system* (MQS) is a restricted form of dissemination quorum system. D-consistency is replaced with the following stronger property: (M-Consistency) $\forall Q_1, Q_2 \in \mathcal{Q},\ \forall B_1, B_2 \in \mathcal{B} : (Q_1 \cap Q_2) \setminus B_1 \not\subseteq B_2$ This guarantees that any two quorums intersect over $2f+1$ processes, and therefore over (at least) $f+1$ correct processes. One type of MQS is a grid quorum system. It consists of a column together with $2f+1$ rows. Figure 4b depicts such a system when $f = 2$. Note that Figure 4a illustrates also a masking quorum, with $f = 20$. This system [50] consists of all $Q \subset \mathbb{P}$ with $|Q| = \lceil \frac{n+2f+1}{2} \rceil$.

Dissemination and masking quorum systems are used in several works to construct (safe and atomic) registers [70]. Figure 4c illustrates a novel type of Byzantine quorums systems called *witnessing quorum system* (WQS). A WQS is a special case of masking quorum system. In detail,



(a) (b) (c)

Figure 4: Illustrating quorums in (a) DQS, (b) MQS and (c) WQS quorum systems.

recall that $\mathcal{Q}$ denotes the quorum system and $\mathcal{B}$ the adversary. WQS replaces the M-Consistency property of MQS with the following one: (W-Consistency) $\forall Q_1, Q_2 \in \mathcal{Q},\ \forall B_1, B_2, B_3 \in \mathcal{B} : (Q_1 \cap Q_2) \setminus (B_1 \cup B_2) \not\subseteq B_3$ 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.

An example of WQS is illustrated in Figure 4c. This quorum system is built as a variation of the *M-grid quorum* system proposed in [51]. 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 in 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 4c, 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 S-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 $\mathfrak{Q}$ belongs to $O(\sqrt{f/n})$ when quorums are picked uniformly at random.*

Proof. Let $c(\mathfrak{Q})$ be the size of the smallest quorum in the S-BQS quorum system $\mathfrak{Q}$. From [50], we know that the load of $\mathfrak{Q}$ is $c(\mathfrak{Q})/n$. All the quorums in $\mathfrak{Q}$ system are of the same size, that is $2 \times \zeta \times \sqrt{n} - \zeta^2$. Hence, the load of $\mathfrak{Q}$ is given by: $(2 \times \zeta \times \sqrt{n} - \zeta^2)/n \in O(\sqrt{f/n})$ □

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: $\boxed{f}_{j \in J} l_j = \sqcup \{e : count(e) \geq f+1\}$, where $count(e) = |\{j \in J : e \sqsubseteq l_j\}|$. In other words, the operator $\boxed{f}_{j \in J} l_j$ 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 $\lfloor f \rfloor_{j \in J}\ deps_j$ are all the commands reported at least f times. In which case, $(\mathcal{L}, \sqsubseteq)$ equals $(2^{\mathcal{C}}, \subseteq)$. This is the original usage in [33].

Another example is provided in §5.2.1, where we use compaction over dependencies. In that case, $\mathcal{L}$ are all the vectors in $\mathbb{P} \times \mathbb{N}$. 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.

In §5.3, we use the threshold union operator to compute a timestamp. Here, $(\mathcal{L}, \sqsubseteq)$ is set to $(\mathbb{N}, <)$.

D Proofs

D.1 Correctness of Algorithm 1 (BLSMR Machine)

This section establishes the correctness of Algorithm 1. Let r be a run of the algorithm. Our first result characterizes the necessary trust to execute successfully the assignment in line 16.

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

Proof. From the definition of $1_{\mathcal{C}}$ (§4.1) and because commands are non-forgeable (see the system model in §2.1) if $1_{\mathcal{C}}(\mathsf{c})$ evaluates to *true* then c must be a state-machine command. □

Proposition 20. *Consider that a correct process p executes in r the trusted assignment in line 16, 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 command,* (ii) *DDS is called with input c and returns (D, b) in r, and* (iii) *either $b = true$, or the service $\mathsf{CONS}_{\mathsf{c}}$ 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. First of all, assume that $b = true$. At the light of the definition of $check_v$, $check_{\mathsf{DDS}}(\mathsf{c}, (D, b))$ equals *true*. Service DDS is trusted wrt. $(1_{\mathcal{C}}, \_)$. As a consequence, DDS is verifying wrt. $(1_{\mathcal{C}}, \_)$. Applying Proposition 19, c is a command. Since $check_{\mathsf{DDS}}(\mathsf{c}, (D, b))$ equals *true* and the DDS service is verifiable, the service is called with input c and returns (D, b) in r. Conversely, assume that b equals *false*. In that case, by the definition of $check_v$, $check_{\mathsf{CONS}_{\mathsf{c}}}(D, D)$ evaluates to *true*. Because $\mathsf{CONS}_{\mathsf{c}}$ is verifiable, it must have returned D in the run r. By the Validity property of Consensus, D is proposed by a process. Since $\mathsf{CONS}_{\mathsf{c}}$ is trusted wrt. $(\mathsf{DDS}, \mathsf{c})$ and $propose(D) = D$ occurs in r, necessarily $check_{\mathsf{DDS}}(c, (D, \_)) = true$. Then, because DDS is trusted wrt. $(1_{\mathcal{C}}, \_)$, the command c is submitted in r and DDS was called with input c and returned $(D, \_)$ during the run. □

Theorem 2. Algorithm 1 implements BLSMR.

Proof. We prove that all the properties of BLSMR hold.

(Validity) If some command c is committed at a correct process p, then $D = deps(\mathsf{c}) \neq \bot$ holds at p. Necessarily p executes line 16 for c. Applying Proposition 20, c is submitted and $announce(\mathsf{d})$ returns $(D, \_)$. By validity of the DDS service, for any $\mathsf{d} \in D$, d is a command.

(Consistency) Let c and d be two conflicting committed commands at some correct process p. By definition, the two commands are committed at p when $D = deps(\mathsf{c})$, $D' = deps(\mathsf{d})$ with $D, D' \in 2^{\mathcal{C}}$. Applying Proposition 20, we know that DDS returns D and D' when called respectively with command c and d in r. 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, $\mathsf{DDS}.announce(\mathsf{c})$ returns (D, b) in r. Similarly, from the fact that q commits value D' for $deps(\mathsf{c})$, $\mathsf{DDS}.announce(\mathsf{c})$ returns (D', b') in r. 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) We prove that once submitted at a correct process p, a command c always ends up being 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 show that one such tuple exists.

Both CONS and DDS are live services, that is a calll at a correct process returns. It follows that the calls in lines 9 and 10 return at process p. Thus, p eventually executes line 11. Since links are reliable, process q eventually receives a tuple (c, D, b) from p (line 13).

From what preceeds, process q receives a tuple (c, D, b), from p. If b equals *true*, then (D, b) is the value p returned at line 9 from calling $\mathsf{DDS}.announce(\mathsf{c})$. Since DDS is verifiable, $check_{\mathsf{DDS}}(\mathsf{c}, (D, b))$ evaluates to *true* at q. Hence, $check_v((\mathsf{c}, D, b, D)$ 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 10. The Validity property of consensus implies that D is proposed by some process. Since $\mathsf{CONS_c}$ is verifiable, $check_{\mathsf{CONS_c}}(D, D)$ evaluates to *true* at q. It follows that the assignment in line 16 also succeeds in this case.

□

D.2 Correctness of Algorithm 2 (à la EPaxos)

Proposition 1. Algorithm 2 implements a trusted DDS service.

Proof. Consider some run r of Algorithm 2.

(Validity) Consider that Algorithm 2 returns $(D, \_)$. D is computed in line 6, as the union of $(D_j)_{j \in Q}$ for some quorum Q. For each process $j \in Q$, line 6 verifies that $D_j \subseteq \mathcal{C}$. Pick $\mathsf{d} \in D$. Since commands are non-forgeable by replicas, d is a command.

(Visibility) Let c and d be two conflicting commands announced in r. Let D and D' be respectively the response to $announce_p(\mathsf{c})$ and $announce_q(\mathsf{d})$, with p and q two correct processes. The dependencies of a command are returned at line 9 and computed at line 7 through the execution of the union operator. Consider the execution of this line by p. The union operator is parameterized with some quorum Q_{c}, obtained at line 6. Similarly, let Q_{d} be the quorum used by q. From the D-Consistency property of the *BQS* service, the quorums Q_{c} and Q_{d} intersect over at least $f + 1$ correct processes. Hence, at least one correct process, say j, executes lines 10 to 13 for the two commands. Without lack of generalities, 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 process q with a set including c. According to line 7, $\mathsf{c} \in D'$ holds when q returns it in line 9.

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

(***Trust***) Next, we explain how the *check* function (line 15) ensures that Algorithm 2 performs correctly on some input c. This function consists of a series of safety verifications in order to bound the damage a Byzantine announcer would make to the service. At line 17, the function first checks that Q is a proper quorum. Further, it verifies that each of the reported dependency is computed by the corresponding process in the quorum (line 18). At line 19, the function re-calculates the union of dependencies, to guarantee that the announcer did not omit anything. Further, in line 20, it verifies that the fast path is forbidden. Finally, it returns *true* only if all of the above verifications succeed, ensuring that Algorithm 2 is verifiable.

A correct process also verifies that c is a command before replying to the announcer (line 12). Hence, Algorithm 2 is verifying wrt. $(1_{\mathcal{C}}, \_)$, as required.

(Liveness) Let B be the (Byzantine) failure pattern of r and p a correct process announcing a command. By the D-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 correct 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.

□

D.3 Correctness of Algorithm 3 (Wintermute)

Lemma 21. *If a correct process executes reply(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*(c) for the first timen, 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, c 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 that Algorithm 3 returns $(D, \_)$. D is computed in line 7, as the threshold union of $(D_j)_{j \in Q}$ for some quorum Q. Pick $\mathsf{d} \in \lfloor f \rfloor_{j \in Q} D_j$. By definition, 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 is a command.

(Visibility) Let a and b be two conflicting commands submitted in run r. Assume that two correct processes p and q return $(D, \_)$ and $(D', \_)$ after calling respectively *announce*(a) and *announce*(b) in r.

The dependencies of a command are returned at line 9 and calculated in line 7 using the threshold union operator. Consider the execution of this line by process p. The threshold union operator is parameterized with a quorum Q_{a}, obtained in line 6. Similarly, let Q_{b} be the quorum obtained by process q.

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

lack of generality, consider that this is the later case that happens during r. For each of these processes j, $\mathsf{b} \in \mathit{deps}_j(\mathsf{a})$ holds. Hence, b is reported $f+1$ times in $(\mathit{deps}_j)_{j \in Q_\mathsf{a}}$. Consequently, applying the definition of the threshold union operator, we have $\mathsf{b} \in D$, as required.

(Weak agreement) Consider that during run r two correct processes p and p' returns respectively (D, true) and $(D', \_)$. A dependency set is computed in line 7. Let Q and Q' be the two quorums used at line 6 by p and p', respectively. ($D \subseteq D'$) Let c be some command in D. According to line 25, $\mathsf{c} \in D_j$ for any process $j \in Q$. In Algorithm 3, variable BQS is a witnessing quorum system. There are $n \geq 3f+1$ processes in the system and $Q = \mathbb{P}$. As a consequence, Q and Q' intersect over $2f+1$ correct processes. At line 10, $f+1$ of these processes receive a message from p before they receive a message from p'. By Lemma 21, because they are all correct and report command c to p, they also report command c to p'. Since c appears $f+1$ times in $(D_j)_{j \in Q'}$, it is included in $\underline{\lfloor f \rfloor}_{j \in Q'} D_j$. Thus, the dependency set D' computed by p' contains command c. ($D' \subseteq D$) Consider some command $\mathsf{c} \in D'$. Command c is reported by $f+1$ processes (line 7). 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 set dependency set. The predicate at line 25 requires that $D = D_j$. It follows that $\mathsf{c} \in D$.

(Trust) This is analogous to the proof provided in Proposition 1. The main difference is that now at line 19, the verifier calculates the threshold union of dependencies instead of an union.

(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).

□

Theorem 3. Wintermute is a fast 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. It remains to show that the protocol is fast. Suppose a conflict-free command c submitted at a process p during a run r. Command c is announced at line 9 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. By Validity, d conflicts with c. Assume that q is the process that submitted d in r. Since c is conflict-free and all the processes are correct, d is executed everywhere at the time c, or the converse hold. Suppose the later case. Because $\mathsf{d} \in D$, some process q returns $\mathsf{d} \in \mathit{deps}(\mathsf{c})$ at line 13 in Algorithm 1. At the time d is submitted, c is already executed at q. This implies that c precedes d in the log of q. (Recall that a log contains (at most) a single occurence of each command.) Contradiction. From what precedes, every command in the committed dependencies of c is already executed at every process at the time c is submitted. Hence, all the processes return the same set of dependencies. It follows that c is stable.

□

D.4 Correctness of Algorithm 4 (3Jane)

Proposition 3. Algorithm 4 implements a trusted DDS service.

Proof. We show that all the expected properties hold.

(Validity) Consider that Algorithm 4 returns $(t, \_)$ at some time τ. Timestamp t is computed in line 8, as the threshold union of $(t_j)_{j \in Q}$ for some quorum Q. Pick $\mathsf{d} \in \mathit{deps}(\mathsf{c})$. By definition, for some $Q' \in \mathit{BQS.quorums}()$, $\lfloor f \rfloor_{j \in Q'} \mathit{promises}(\mathsf{d})_j < t$. That is, for any $P \subseteq Q'$, if $|P| \geq f + 1$ then $\mathit{promises}(\mathsf{d})_j < t$ for some $j \in P$. Because BQS is a masking quorum, $|Q \cap Q'| \geq 2f + 1$. Let C be the correct processes in this intersection. There are at least $f + 1$ of these. Due to line 20, for any $j \in C$, $\mathit{promises}[j][\mathsf{c}] \neq 0$. By definition of $\mathit{promises}(\mathsf{d})_j$, either $\mathit{promises}[j][\mathsf{d}] \neq 0$, or $\mathit{promises}(\mathsf{d})_j > \mathit{promises}[j][\mathsf{c}]$, at process p. If for some j the former case holds, necessarily $\mathit{announce}(\mathsf{d})$ was invoked earlier. Otherwise, $\lfloor f \rfloor_{j \in C} \mathit{promises}(\mathsf{d})_j > t$; a contradiction. (Ties are treated similarily.)

(Visibility) Consider c and d two commands. Assume that two processes p and q return $(t, \_)$ and $(t', \_)$ when announcing respectively c and d. Without lack of generality, suppose that $t' < t$. Consider the point in time τ where process p returns from the announcement of c. We note D the dependencies of command c inferred from t. This set contains all the commands c' such that for some $Q' \in \mathit{BQS.quorums}()$, $\lceil f \rceil_{j \in Q'} \mathit{promises}(\mathsf{c}')_j < t$ at time τ. In what follows, we prove the existence of such a quorum Q' for command d.

Command d is announced by process q with timestamp t'. Let Q' be the quorum used by q at line 7 in Algorithm 4, with $t' = \lfloor f \rfloor_{j \in Q'} t_j$. For every correct process $j \in Q'$, if process p updates $\mathit{promises}[j][\mathsf{d}]$ at line 21, then it equals t_j. It follows that at time τ, for every such j, $\mathit{promises}[j][\mathsf{d}] \in \{0, t_j\}$ at p. Then, pick any $f + 1$ correct processes $P \subseteq Q'$. By definition of t', for some process $j \in P$, $\mathit{promises}[j][\mathsf{d}] \leq t'$.

We show that $\lceil f \rceil_{j \in Q'} \mathit{promises}(\mathsf{d})_j \leq t'$ at time τ. By definition, $\lceil f \rceil_{j \in Q'} \mathit{promises}(\mathsf{d})_j = \sqcup\{\hat{t} : \mathit{count}(\hat{t}) \geq 2f + 1\}$, where $\mathit{count}(\hat{t}) = |\{j \in Q' : \hat{t} \leq \mathit{promises}(\mathsf{d})_j\}|$. Pick the largest $\hat{t}$ satisfying $\hat{t} \leq \mathit{promises}(\mathsf{d})_j$ for all $j \in P$, with $P \subseteq Q'$ and $|P| \geq 2f + 1$. Among these processes, at least $f + 1$ are correct; name them P. From what precedes, for some $j \in P$, $\mathit{promises}[j][\mathsf{d}] \leq t'$. Hence, as required, $\hat{t} < t'$.

(Weak agreement) This property is immediate since a call to *announce* always returns $(\_, \mathit{false})$ at a correct process.

(Trust) The verifier executes a series of checks to guarantee that the coordinator did not misbehave. In detail, it first checks that Q is a witnessing Byzantine quorum (line 24) In line 25, it verifies that for every process $j \in Q$, t_j is indeed a timestamp computed by j. Further, in line 26, the verifier re-calculates the threshold union. The last check is to ensure that the fast path is forbidden, after which the verifier returns (line 28).

(Liveness) The proof follows the same logic as with the other solutions, that is Algorithms 2 and 3.

□

Proposition 4. Algorithm 4 with a fast path is trusted DDS service.

Proof. The proof is as follows.

(Validity/Visibility) In the fast-path variation, BQS returns witnessing Byzantine quorums. Because W-Consistency is stricter than M-Consistency, a witnessing quorum system is also a masking quorum system. Hence, we may use the same proofs as with the vanilla version to show that the Validity and Visibility properties hold.

(Weak agreement) Pick a command c and let Q be the quorum $\mathit{BQS.getQuorum}(\mathsf{c})$. Assume that $\mathit{coord}(\mathsf{c})$ returns (t, true) and that process p recovers c with (t', false). Every process in Q replies $t = \lfloor f \rfloor_{j \in Q} t_j$. Note $(t'_j)_{j \in P}$ the replies received by p. By construction, P

is a subset of Q that contains $|Q| - f$ processes such that $t' = \lfloor f \rfloor_{j \in P} t'_j$. By definition, $\lfloor f \rfloor_{j \in P} t'_j = \sqcup\{\hat{t} : count(\hat{t}) \geq f + 1\}$, where $count(\hat{t}) = |\{j \in P : \hat{t} \leq t'_j\}|$. Because $coord(\mathsf{c})$ takes the fast path, it must be the case that $t'_j = t$, with $j \in P$ correct. As a consequence, only timestamp t can be reported $f + 1$ times by the processes in P. It follows that $t' = t$.

(Trust) If the fast path is not taken, then the same verifier as in the original algorithm is used. Otherwise, the fast-path condition is also verified.

(Liveness) We establish that a recovery always succeeds in announcing a command c. Let Q be the quorum $BQS.getQuorum(\mathsf{c})$. Consider a run r with B denoting the faulty processes. There are at most f processes in B. Hence, $P = Q \setminus B$ is responsive. As a consequence, during the recovery of command c all the processes in P reply. It follows that the announcement of command c returns.

□

Theorem 4. 3Jane is a fast state-machine replication protocol.

Proof. The proof follows a similar logic to the one of Theorem 3. □